IAS. 31

METHOD AND ARTICLE OF MANUFACTURE FOR COMPONENT BASED TASK HANDLING DURING CLAIM PROCESSING.

5

FIELD OF THE INVENTION

The present invention relates to task management and more particularly to handling task during insurance claim processing utilizing a computer system.

10

BACKGROUND OF THE INVENTION

Computers have become a necessity in life today. They appear in nearly every office and household worldwide. A representative hardware environment is

15 depicted in prior art Figure 1, which illustrates a typical hardware configuration of a workstation having a central processing unit 110, such as a microprocessor, and a number of other units interconnected via a system bus 112. The workstation shown in Figure 1 includes a Random Access Memory (RAM) 114, Read Only Memory (ROM) 116, an I/O adapter 118 for connecting peripheral

20 devices such as disk storage units 120 to the bus 112, a user interface adapter 122 for connecting a keyboard 124, a mouse 126, a speaker 128, a microphone 132, and/or other user interface devices such as a touch screen (not shown) to the bus 112, communication adapter 134 for connecting the workstation to a communication network (e.g., a data processing network) and a display adapter

25 136 for connecting the bus 112 to a display device 138. The workstation typically has resident thereon an operating system such as the Microsoft Windows NT or Windows/95 Operating System (OS), the IBM OS/2 operating system, the MAC OS, or UNIX operating system.

30 Object oriented programming (OOP) has become increasingly used to develop complex applications. As OOP moves toward the mainstream of software design

Object oriented programming (OOP) has become increasingly used to develop complex applications. As OOP moves toward the mainstream of software design and development, various software solutions require adaptation to make use of the benefits of OOP. A need exists for these principles of OOP to be applied to a

5   messaging interface of an electronic messaging system such that a set of OOP classes and objects for the messaging interface can be provided.

OOP is a process of developing computer software using objects, including the steps of analyzing the problem, designing the system, and constructing the

10   program. An object is a software package that contains both data and a collection of related structures and procedures. Since it contains both data and a collection of structures and procedures, it can be visualized as a self-sufficient component that does not require other additional structures, procedures or data to perform its specific task. OOP, therefore, views a computer program as a collection of

15   largely autonomous components, called objects, each of which is responsible for a specific task. This concept of packaging data, structures, and procedures together in one component or module is called encapsulation.

In general, OOP components are reusable software modules which present an

20   interface that conforms to an object model and which are accessed at run-time through a component integration architecture. A component integration architecture is a set of architecture mechanisms which allow software modules in different process spaces to utilize each others capabilities or functions. This is generally done by assuming a common component object model on which to

25   build the architecture. It is worthwhile to differentiate between an object and a class of objects at this point. An object is a single instance of the class of objects, which is often just called a class. A class of objects can be viewed as a blueprint, from which many objects can be formed.

30   OOP allows the programmer to create an object that is a part of another object. For example, the object representing a piston engine is said to have a

composition-relationship with the object representing a piston. In reality, a piston engine comprises a piston, valves and many other components; the fact that a piston is an element of a piston engine can be logically and semantically represented in OOP by two objects.

5

OOP also allows creation of an object that "depends from" another object. If there are two objects, one representing a piston engine and the other representing a piston engine wherein the piston is made of ceramic, then the relationship between the two objects is not that of composition. A ceramic piston engine does

10   not make up a piston engine. Rather it is merely one kind of piston engine that has one more limitation than the piston engine; its piston is made of ceramic. In this case, the object representing the ceramic piston engine is called a derived object, and it inherits all of the aspects of the object representing the piston engine and adds further limitation or detail to it. The object representing the

15   ceramic piston engine "depends from" the object representing the piston engine. The relationship between these objects is called inheritance.

When the object or class representing the ceramic piston engine inherits all of the aspects of the objects representing the piston engine, it inherits the thermal

20   characteristics of a standard piston defined in the piston engine class. However, the ceramic piston engine object overrides these ceramic specific thermal characteristics, which are typically different from those associated with a metal piston. It skips over the original and uses new functions related to ceramic pistons. Different kinds of piston engines have different characteristics, but may

25   have the same underlying functions associated with it (e.g., how many pistons in the engine, ignition sequences, lubrication, etc.). To access each of these functions in any piston engine object, a programmer would call the same functions with the same names, but each type of piston engine may have different/overriding implementations of functions behind the same name. This

30   ability to hide different implementations of a function behind the same name is called polymorphism and it greatly simplifies communication among objects.

With the concepts of composition-relationship, encapsulation, inheritance and polymorphism, an object can represent just about anything in the real world. In fact, the logical perception of the reality is the only limit on determining the kinds

5   of things that can become objects in object-oriented software. Some typical categories are as follows:

- Objects can represent physical objects, such as automobiles in a traffic-flow simulation, electrical components in a circuit-design program, countries in an economics model, or aircraft in an air-traffic-control

10   system.

- Objects can represent elements of the computer-user environment such as windows, menus or graphics objects.

- An object can represent an inventory, such as a personnel file or a table of the latitudes and longitudes of cities.

15 - An object can represent user-defined data types such as time, angles, and complex numbers, or points on the plane.

With this enormous capability of an object to represent just about any logically separable matters, OOP allows the software developer to design and implement a

20   computer program that is a model of some aspects of reality, whether that reality is a physical entity, a process, a system, or a composition of matter. Since the object can represent anything, the software developer can create an object which can be used as a component in a larger software project in the future.

25   If 90% of a new OOP software program consists of proven, existing components made from preexisting reusable objects, then only the remaining 10% of the new software project has to be written and tested from scratch. Since 90% already came from an inventory of extensively tested reusable objects, the potential domain from which an error could originate is 10% of the program. As a result,

OOP enables software developers to build objects out of other, previously built objects.

This process closely resembles complex machinery being built out of assemblies
5   and sub-assemblies. OOP technology, therefore, makes software engineering more like hardware engineering in that software is built from existing components, which are available to the developer as objects. All this adds up to an improved quality of the software as well as an increased speed of its development.

10

15

-6-

SUMMARY OF THE INVENTION

5   A computer program is provided for developing component based software capable of handling insurance-related tasks. The program includes a data component that stores, retrieves and manipulates data utilizing a plurality of functions. Also provided is a client component which includes an adapter component that transmits and receives data to/from the data component. The

10   client component also includes a business component that serves as a data cache and includes logic for manipulating the data. A controller component is also included which is adapted to handle events generated by a user utilizing the business component to cache data and the adapter component to ultimately persist data to a data repository. In use, the client component allows a user to define

15   tasks that achieve an insurance-related goal upon completion. In addition, the user is able to input rules which dictate which tasks should be selected based on a set of predetermined events. Events are then received from any source, such as a common event queue. Finally, tasks are selected and outputted based on the received events.

20

25

DESCRIPTION OF THE DRAWINGS

The foregoing and other objects, aspects and advantages are better understood
from the following detailed description of a preferred embodiment of the
invention with reference to the drawings, in which:

Prior Art Figure 1 is a schematic diagram of the present invention; and

Figure 2A is block diagram of one embodiment of the present invention.

Figure 2B is a flowchart showing how components generally operate in
accordance with one embodiment of the present invention.

Figure 2C is a flowchart showing how the UI Controller operates in accordance
with one embodiment of the present invention.

Figure 2D is a flowchart showing the interactions between the CCA, the CCI, and
the Server Component in accordance with one embodiment of the present
invention.

Figure 3 shows the life cycle of a typical User Interface and the standard methods
that are part of the Window Processing Framework.

Figure 4 is an illustration showing how different languages are repainted and
recompiled.

Figure 5 is a block diagram of an Architecture Object.

Figure 6 is an illustration showing the physical layout of CodeDecode tables
according to one embodiment of the present invention.

Figure **7** is a logic diagram according to one embodiment of the present invention.

5   Figure **8** is a block diagram of the security framework and its components.

Figure **9** is an illustration showing the relationships between the security element and other elements.

10   Figure **10** is an illustration of the Negotiation component of one embodiment of the present invention;

Figure **11** is a flow diagram of the operations carried out by the Organization component of one embodiment of the present invention;

15

Figure **12** is an illustration of the Participant component of one embodiment of the present invention;

Figure **13** is a flow diagram of the operations carried out by the Task Assistant
20   component of one embodiment of the present invention;

Figure **14** is an illustration of the Event Processor in combination with other components of the system in accordance with on embodiment of the present invention; and

25

Figure **15** is an illustration of the Task Engine in accordance with one embodiment of the present invention.

DISCLOSURE OF THE INVENTION

5   Programming languages are beginning to fully support the OOP principles, such
as encapsulation, inheritance, polymorphism, and composition-relationship. With
the advent of the C++ language, many commercial software developers have
embraced OOP. C++ is an OOP language that offers a fast, machine-executable
code. Furthermore, C++ is suitable for both commercial-application and
10   systems-programming projects. For now, C++ appears to be the most popular
choice among many OOP programmers, but there is a host of other OOP
languages, such as Smalltalk, Common Lisp Object System (CLOS), and Eiffel.
Additionally, OOP capabilities are being added to more traditional popular
computer programming languages such as Pascal.

15

The benefits of object classes can be summarized, as follows:

- Objects and their corresponding classes break down complex
  programming problems into many smaller, simpler problems.
- Encapsulation enforces data abstraction through the organization of data
20   into small, independent objects that can communicate with each other.
  Encapsulation protects the data in an object from accidental damage, but
  allows other objects to interact with that data by calling the object's
  member functions and structures.
- Subclassing and inheritance make it possible to extend and modify objects
25   through deriving new kinds of objects from the standard classes available
  in the system. Thus, new capabilities are created without having to start
  from scratch.
- Polymorphism and multiple inheritance make it possible for different
  programmers to mix and match characteristics of many different classes
30   and create specialized objects that can still work with related objects in
  predictable ways.

- Class hierarchies and containment hierarchies provide a flexible mechanism for modeling real-world objects and the relationships among them.

- Libraries of reusable classes are useful in many situations, but they also have some limitations. For example:

- Complexity. In a complex system, the class hierarchies for related classes can become extremely confusing, with many dozens or even hundreds of classes.

- Flow of control. A program written with the aid of class libraries is still responsible for the flow of control (i.e., it must control the interactions among all the objects created from a particular library). The programmer has to decide which functions to call at what times for which kinds of objects.

- Duplication of effort. Although class libraries allow programmers to use and reuse many small pieces of code, each programmer puts those pieces together in a different way. Two different programmers can use the same set of class libraries to write two programs that do exactly the same thing but whose internal structure (i.e., design) may be quite different, depending on hundreds of small decisions each programmer makes along the way. Inevitably, similar pieces of code end up doing similar things in slightly different ways and do not work as well together as they should.

Class libraries are very flexible. As programs grow more complex, more programmers are forced to reinvent basic solutions to basic problems over and over again. A relatively new extension of the class library concept is to have a framework of class libraries. This framework is more complex and consists of significant collections of collaborating classes that capture both the small scale patterns and major mechanisms that implement the common requirements and design in a specific application domain. They were first developed to free application programmers from the chores involved in displaying menus,

windows, dialog boxes, and other standard user interface elements for personal computers.

Frameworks also represent a change in the way programmers think about the
5   interaction between the code they write and code written by others. In the early days of procedural programming, the programmer called libraries provided by the operating system to perform certain tasks, but basically the program executed down the page from start to finish, and the programmer was solely responsible for the flow of control. This was appropriate for printing out paychecks,
10   calculating a mathematical table, or solving other problems with a program that executed in just one way.

The development of graphical user interfaces began to turn this procedural programming arrangement inside out. These interfaces allow the user, rather than
15   program logic, to drive the program and decide when certain actions should be performed. Today, most personal computer software accomplishes this by means of an event loop which monitors the mouse, keyboard, and other sources of external events and calls the appropriate parts of the programmer's code according to actions that the user performs. The programmer no longer
20   determines the order in which events occur. Instead, a program is divided into separate pieces that are called at unpredictable times and in an unpredictable order. By relinquishing control in this way to users, the developer creates a program that is much easier to use. Nevertheless, individual pieces of the program written by the developer still call libraries provided by the operating
25   system to accomplish certain tasks, and the programmer must still determine the flow of control within each piece after it's called by the event loop. Application code still "sits on top of" the system.

Even event loop programs require programmers to write a lot of code that should
30   not need to be written separately for every application. The concept of an application framework carries the event loop concept further. Instead of dealing

with all the nuts and bolts of constructing basic menus, windows, and dialog boxes and then making these things all work together, programmers using application frameworks start with working application code and basic user interface elements in place. Subsequently, they build from there by replacing some of the generic capabilities of the framework with the specific capabilities of the intended application.

Application frameworks reduce the total amount of code that a programmer has to write from scratch. However, because the framework is really a generic application that displays windows, supports copy and paste, and so on, the programmer can also relinquish control to a greater degree than event loop programs permit. The framework code takes care of almost all event handling and flow of control, and the programmer's code is called only when the framework needs it (e.g., to create or manipulate a proprietary data structure).

A programmer writing a framework program not only relinquishes control to the user (as is also true for event loop programs), but also relinquishes the detailed flow of control within the program to the framework. This approach allows the creation of more complex systems that work together in interesting ways, as opposed to isolated programs, having custom code, being created over and over again for similar problems.

Thus, as is explained above, a framework basically is a collection of cooperating classes that make up a reusable design solution for a given problem domain. It typically includes objects that provide default behavior (e.g., for menus and windows), and programmers use it by inheriting some of that default behavior and overriding other behavior so that the framework calls application code at the appropriate times.

There are three main differences between frameworks and class libraries:

• Behavior versus protocol. Class libraries are essentially collections of behaviors that you can call when you want those individual behaviors in your program. A framework, on the other hand, provides not only behavior but also the protocol or set of rules that govern the ways in which behaviors can be combined, including rules for what a programmer is supposed to provide versus what the framework provides.

• Call versus override. With a class library, the code the programmer instantiates objects and calls their member functions. It's possible to instantiate and call objects in the same way with a framework (i.e., to treat the framework as a class library), but to take full advantage of a framework's reusable design, a programmer typically writes code that overrides and is called by the framework. The framework manages the flow of control among its objects. Writing a program involves dividing responsibilities among the various pieces of software that are called by the framework rather than specifying how the different pieces should work together.

• Implementation versus design. With class libraries, programmers reuse only implementations, whereas with frameworks, they reuse design. A framework embodies the way a family of related programs or pieces of software work. It represents a generic design solution that can be adapted to a variety of specific problems in a given domain. For example, a single framework can embody the way a user interface works, even though two different user interfaces created with the same framework might solve quite different interface problems.

Thus, through the development of frameworks for solutions to various problems and programming tasks, significant reductions in the design and development effort for software can be achieved. A preferred embodiment of the invention utilizes HyperText Markup Language (HTML) to implement documents on the Internet together with a general-purpose secure communication protocol for a transport medium between the client and the Newco. HTTP or other protocols

could be readily substituted for HTML without undue experimentation. Information on these products is available in T. Berners-Lee, D. Connoly, "RFC 1866: Hypertext Markup Language - 2.0" (Nov. 1995); and R. Fielding, H, Frystyk, T. Berners-Lee, J. Gettys and J.C. Mogul, "Hypertext Transfer Protocol

5   -- HTTP/1.1: HTTP Working Group Internet Draft" (May 2, 1996). HTML is a simple data format used to create hypertext documents that are portable from one platform to another. HTML documents are SGML documents with generic semantics that are appropriate for representing information from a wide range of domains. HTML has been in use by the World-Wide Web global information

10   initiative since 1990. HTML is an application of ISO Standard 8879; 1986 Information Processing Text and Office Systems; Standard Generalized Markup Language (SGML).

To date, Web development tools have been limited in their ability to create

15   dynamic Web applications which span from client to server and interoperate with existing computing resources. Until recently, HTML has been the dominant technology used in development of Web-based solutions. However, HTML has proven to be inadequate in the following areas:

- Poor performance;

20   - Restricted user interface capabilities;

- Can only produce static Web pages;

- Lack of interoperability with existing applications and data; and

- Inability to scale.

25   Sun Microsystem's Java language solves many of the client-side problems by:

- Improving performance on the client side;

- Enabling the creation of dynamic, real-time Web applications; and

- Providing the ability to create a wide variety of user interface components.

30

With Java, developers can create robust User Interface (UI) components. Custom "widgets" (e.g., real-time stock tickers, animated icons, etc.) can be created, and client-side performance is improved. Unlike HTML, Java supports the notion of client-side validation, offloading appropriate processing onto the client for

5   improved performance. Dynamic, real-time Web pages can be created. Using the above-mentioned custom UI components, dynamic Web pages can also be created.

Sun's Java language has emerged as an industry-recognized language for

10   "programming the Internet." Sun defines Java as: "a simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high-performance, multithreaded, dynamic, buzzword-compliant, general-purpose programming language. Java supports programming for the Internet in the form of platform-independent Java applets." Java applets are small, specialized

15   applications that comply with Sun's Java Application Programming Interface (API) allowing developers to add "interactive content" to Web documents (e.g., simple animations, page adornments, basic games, etc.). Applets execute within a Java-compatible browser (e.g., Netscape Navigator) by copying code from the server to client. From a language standpoint, Java's core feature set is based on

20   C++. Sun's Java literature states that Java is basically, "C++ with extensions from Objective C for more dynamic method resolution."

Another technology that provides similar function to JAVA is provided by Microsoft and ActiveX Technologies, to give developers and Web designers

25   wherewithal to build dynamic content for the Internet and personal computers. ActiveX includes tools for developing animation, 3-D virtual reality, video and other multimedia content. The tools use Internet standards, work on multiple platforms, and are being supported by over 100 companies. The group's building blocks are called ActiveX Controls, small, fast components that enable

30   developers to embed parts of software in hypertext markup language (HTML) pages. ActiveX Controls work with a variety of programming languages

including Microsoft Visual C++, Borland Delphi, Microsoft Visual Basic programming system and, in the future, Microsoft's development tool for Java, code named "Jakarta." ActiveX Technologies also includes ActiveX Server Framework, allowing developers to create server applications. One of ordinary

5   skill in the art readily recognizes that ActiveX could be substituted for JAVA without undue experimentation to practice the invention.

Detailed Description

10

One embodiment of the present invention is a server based framework utilizing component based architecture. Referring to Figure 2A, one embodiment of the present invention includes an Architecture Object 200, an Application Object 202, a User Interface Form 204, a User Interface Controller 206, a Client

15   Component Adapter 208, a COM Component Interface 210, and a Server Component 222.

In general, the components of the present invention operate as shown in Figure 2B. In step 230, data is stored in an object of the component. In step 232,

20   functions which manipulate the object are encapsulated with the object data. Later, in step 234, the stored object data can be manipulated by other components utilizing the functions of step 232.

Architecture Object

25   The Architecture Object 200 provides an easy-to-use object model that masks the complexity of the architecture on the client. The Architecture Object 200 provides purely technical services and does not contain any business logic or functional code. It is used on the client as the single point of access to all architecture services.

30

On the server side, the Architecture Object 200 is supplemented by a set of global functions contained in standard VB modules

The Architecture Object 200 is responsible for providing all client architecture services (i.e., codes table access, error logging, etc.), and a single point of entry for architecture services. The Architecture Object 200 is also responsible for

5 allowing the architecture to exist as an autonomous unit, thus allowing internal changes to be made to the architecture with minimal impact to application.

The Architecture Object 200 provides a code manager, client profile, text manager, ID manager, registry manager, log manager, error manager, and a

10 security manager. The codes manager reads codes from a local database on the client, marshals the codes into objects, and makes them available to the application. The client profile provides information about the current logged-in user. The text manager provides various text manipulation services such as search and replace. The ID manager generates unique IDs and timestamps. The

15 registry manager encapsulates access to the system registry. The log manager writes error or informational messages to the message log. The error manager provides an easy way to save and re-raise an error. And the security manager determines whether or not the current user is authorized to perform certain actions.

20

Application Object

The Application Object 202 has a method to initiate each business operation in the application. It uses late binding to instantiate target UI controllers in order to

25 provide autonomy between windows. This allows different controllers to use the Application Object 202 without statically linking to each and every UI controller in the application.

When opening a UI controller, the Application Object 202 calls the architecture

30 initialization, class initialization, and form initialization member functions.

The Application Object 202 keeps a list of every active window, so that it can shut down the application in the event of an error. When a window closes, it tells the Application Object 202 , and is removed from the Application Object's 202 list of active windows.

5

The Application Object 202 is responsible for instantiating each UI Controller 206, passing data / business context to the target UI Controller 206, and invoking standard services such as initialize controller, initializing Form and Initialize Architecture. The Application Object 202 also keeps track of which windows are

10   active so that it can coordinate the shutdown process.

UI Form

The UI form's 204 primary responsibility is to forward important events to its controller 206. It remains mostly unintelligent and contains as little logic as

15   possible. Most event handlers on the form simply delegate the work by calling methods on the form's controller 206.

The UI form 204 never enables or disables its own controls, but ask its controller 206 to do it instead. Logic is included on the UI form 204 only when it involves

20   very simple field masking or minor visual details.

The UI form 204 presents an easy-to-use, graphical interface to the user and informs its controller 206 of important user actions. The UI form 204 may also provide basic data validation (e.g., data type validation) through input masking.

25   In addition, the UI form is responsible for intelligently resizing itself, launching context-sensitive help, and unload itself.

User Interface Controller

Every UI Controller 206 includes a set of standard methods for initialization,

30   enabling and disabling controls on its UI form 204, validating data on the form, getting data from the UI form 204, and unloading the UI form 204.

UI Controllers 206 contain the majority of logic to manipulate Business Objects 207 and manage the appearance of its UI form 204. If its form is not read-only, the UI Controller 206 also tracks whether or not data on the UI form 204 has

5   changed, so as to avoid unnecessary database writes when the user decides to save. In addition, controllers of auxiliary windows (like the File-Save dialog box in Microsoft Word), keep track of their calling UI controller 206 so that they can notify it when they are ready to close.

10   Figure 2C is a flowchart showing how the UI Controller operates in one embodiment of the present invention. In step 236, data is entered in a UI form by a user. In step 238, the UI controller interprets the data entered into the UI form. In step 240, the UI controller places the appropriate data into a Business Object to be utilized and retrieved later.

15

A UI Controller 206 defines a Logical Unit of Work (LUW). If an LUW involves more than one UI Controller 206, the LUW is implemented as a separate object.

20   The UI Controller 206 is responsible for handling events generated by the user interacting with the UI form 204 and providing complex field validation and cross field validation within a Logical Unit of Work. The UI Controller 206 also contains the logic to interact with business objects 207, and creates new business objects 207 when necessary. Finally, the UI Controller 206 interacts with Client

25   Component Adapters 208 to add, retrieve, modify, or delete business objects 207, and handles all client-side errors.

Business Objects
The Business Object's (BO) 207 primary functionality is to act as a data holder,
30   allowing data to be shared across User Interface Controllers 206 using an object-based programming model.

BOs 207 perform validation on their attributes as they are being set to maintain the integrity of the information they contain. BOs 207 also expose methods other than accessors to manipulate their data, such as methods to change the life cycle
5   state of a BO 207 or to derive the value of a calculated attribute.

In many cases, a BO 207 will have its own table in the database and its own window for viewing or editing operations.

10 Business Objects 207 contain information about a single business entity and maintain the integrity of that information. The BO 207 encapsulates business rules that pertain to that single business entity and maintains relationships with other business objects (e.g., a claim contains a collection of supplements). Finally, the BO 207 provides additional properties relating to the status of the
15 information it contains (such as whether that information has changed or not), provides validation of new data when necessary, and calculates attributes that are derived from other attributes (such as Full Name, which is derived from First Name, Middle Initial, and Last Name).

20 ***Client Component Adapters***

Client Component Adapters (CCAs) 208 are responsible for retrieving, adding, updating, and deleting business objects in the database. CCAs 208 hide the storage format and location of data from the UI controller 206. The UI controller 206 does not care about where or how objects are stored, since this is taken care
25 of by the CCA 208.

The CCA 208 marshals data contained in recordsets returned by the server into business objects 207. CCAs 208 masks all remote requests from UI Controller 206 to a specific component, and act as a "hook" for services such as data
30 compression, and data encryption.

COM Component Interface

A COM Component Interface (CCI) 210 is a "contract" for services provided by a component. By "implementing" an interface (CCI) 210, a component is promising to provide all the services defined by the CCI 20.

5

The CCI 210 is not a physical entity (which is why it is depicted with a dotted line). It's only reason for existence is to define the way a component appears to other objects. It includes the signatures or headers of all the public properties or methods that a component will provide.

10

To implement a CCI 210, a server component exposes a set of specially named methods, one for each method defined on the interface. These methods should do nothing except delegate the request to a private method on the component which will do the real work.

15

The CCI 210 defines a set of related services provided by a component. The CCI allows any component to "hide" behind the interface to perform the services defined by the interface by "implementing" the interface.

20 ### Server Component

Server components 222 are course grained and transaction oriented. They are designed for maximum efficiency.

Server Components 222 encapsulate all access to the database, and define
25 business transaction boundaries. In addition, Server Components 222 are responsible for ensuring that business rules are honored during data access operations.

A Server Component 222 performs data access operations on behalf of CCAs
30 208 or other components and participates in transactions spanning server components 222 by communicating with other server components 222 . The

Server Component 222 is accessible by multiple front end personalities (e.g.,
Active Server Pages), and contains business logic designed to maintain the
integrity of data in the database.

5 Figure 2D is a flowchart showing the interactions between the CCA, the CCI, and
the Server Component in accordance with one embodiment of the present
invention. In step 242, a request is made to place client created data on the server
database. In step 244, the data is transferred to the server component 222
utilizing a CCI 210. In step 246, the server component 222 stores the data in the
10 server database.

BUSINESS RULE PLACEMENT

15 *Overview*

The distribution of business rules across tiers of the application directly affects
the robustness and performance of the system as a whole. Business rules can be
categorized into the following sections: Relationships, Calculations, and Business
Events.

20 ***Relationships between Business Objects***

Business Objects 207 are responsible for knowing other business objects 207
with which they are associated.

Relationships between BOs 207 are built by the CCA 208 during the marshaling
25 process. For example, when a CCA 208 builds a claim BO 207, it will also build
the collection of supplements if necessary.

Calculated Business Data

Business rules involving calculations based on business object 207 attributes are
coded in the business objects 207 themselves. Participant Full Name is a good

example of a calculated attribute. Rather than force the controllers to concatenate the first name, middle initial, and last name every time they wanted to display the full name, a calculated attribute that performs this logic is exposed on the business object. In this way, the code to compose the full name only has to be

5   written once and can be used by many controllers 206.

Another example of a calculated attribute is the display date of a repeating task. When a task with a repeat rule is completed, a new display date must be determined. This display date is calculated based on the date the task was

10   completed, and the frequency of repetition defined by the repeat rule. Putting the logic to compute the new display date into the Task BO 207 ensures that it is coded only once.

Responses to Business Events

Business rules that relate to system events and involve no user interaction are

15   enforced on the server components.

Completion of a task is a major event in the system. When a task is completed, the system first ensures that the performer completing the task is added to the claim. Then, after the task is marked complete in the database, it is checked to

20   see if the task has a repeat rule. If so, another task is created and added to the database. Finally, the event component is notified, because the Task Engine may need to react to the task completion.

Consider the scenario if the logic to enforce this rule were placed on the UI

25   controller 206.

The controller 206 calls the Performer Component to see if the performer completing the task has been added to the claim. If the performer has not been added to the claim, then the controller 206 calls the performer component again

30   to add them.

Next, the controller 206 calls the Task Component to mark the task complete in the database. If the task has a repeat rule, the controller 206 computes the date the task is to be redisplayed and calls the Task Component again to add a new

5   task. Lastly, the controller 206 calls the Event Component to notify the Task Engine of the task completion.

The above implementation requires five network round trips in its worst case. In addition, any other controller 206 or server component 222 that wants to

10   complete a task must code this logic all over again. Enforcing this rule in the task server component 222 reduces the number of network round trips and eliminates the need to code the logic more than once.

Responses to User Events

All responses to user events are coordinated by the controller 206. The controller

15   206 is responsible for actions such as enabling or disabling controls on its form, requesting authorization from the security component, or making calls to the CCA 208.

Authorization

All logic for granting authorization is encapsulated inside the security

20   component. Controllers 206 and components 222 must ask the security component if the current user is authorized to execute certain business operations in the system. The security component will answer yes or no according to some predefined security logic.

Summary

Type of Business Rule	Example	Responsibility
Maintaining relationships between BOs	Claim keeps a collection of supplements	Business Objects
Building relationships between BOs	CCA builds the claim's collection of supplements	CCAs
Calculated Business Data	Participant calculates its full name	Business Objects
Responses to Business	Task Component collaborates with other	Components

Events	components	
Requesting Authorization	Task Library controller asks the security component if the current user is allowed to access Task Library	Controllers and Components
Granting Authorization	Security component determines whether or not the current user can access Task Library	Security Component

WINDOW PROCESSING FRAMEWORK

5

The Default Window Framework provides default window processing for each window contained within the system. This default processing aides the developer in developing robust, maintainable UIs, standardizes common processes (such as form initialization) and facilitates smooth integration with architecture services.

10

Figure 3 shows the life cycle of a typical User Interface and the standard methods that are part of the Window Processing Framework 300.

The Window Processing Framework 300 encompasses the following:

15 Window Initialization 302;

Window Save Processing 304;

Window Control State Management 306;

Window Data Validation 308;

Window Shutdown Processing 310.

20

Window Initialization Processing 302: After creating a controller 206 for the desired window, the App object 202 calls a set of standard initialization functions on the controller 206 before the form 204 is displayed to the user. Standardizing these functions makes the UIs more homogeneous throughout the application,

25 while promoting good functional decomposition.

Window Save Processing 304: Any time a user updates any form text or adds an item to a ListBox, the UI Controller 206 marks the form as "dirty". This allows the UI controller 206 to determine whether data has changed when the form closes and prompt the user to commit or lose their changes.

5

Window Control State Management 306: Enabling and disabling controls and menu options is a very complex part of building a UI. The logic that modifies the state of controls is encapsulated in a single place for maintainability.

10 **Window Data Validation 308:** Whenever data changes on a form, validation rules can be broken. The controller is able to detect those changes, validate the data, and prompt the user to correct invalid entries.

Window Shutdown Processing 310: The Window Shutdown framework
15 provides a clear termination path for each UI in the event of an error. This reduces the chance of memory leaks, and General Protection failures.

20
Benefits

Standardized Processing: Standardizing the window processing increases the homogeneity of the application. This ensures that all windows within the
25 application behave in a consistent manner for the end users, making the application easier to use. It also shortens the learning curve for developers and increases maintainability, since all windows are coded in a consistent manner.

Simplified Development: Developers can leverage the best practices
30 documented in the window processing framework to make effective design and coding decisions. In addition, a shell provides some "canned" code that gives developers a head start during the coding effort.

Layered Architecture: Because several architecture modules provide standardized processing to each application window, the core logic can be changed for every system window by simply making modifications to a single procedure.

5

Window Initialization 302

To open a new window, the App Object 202 creates the target window's controller 206 and calls a series of methods on the controller 206 to initialize it. The calling of these methods, ArchInitClass, InitClass, InitForm, and ShowForm,

10   is illustrated below.

ArchInitClass

The main purpose of the ArchInitClass function is to tell the target controller 206 who is calling it. The App Object 202 "does the introductions" by passing the

15   target controller 206 a reference to itself and a reference to the calling controller 206. In addition, it serves as a hook into the controller 206 for adding architecture functionality in the future.

```
Public Sub ArchInitClass(objApp As Object, objCallingCTLR As
20   Object)
         ' remember who called me
         Set m_objApp = objApp
         Set m_objCallingCTLR = objCallingCTLR
     End Sub
```

25   ### *InitClass*

This function provides a way for the App Object 202 to give the target controller 206 any data it needs to do its processing. It is at this point that the target controller 206 can determine what "mode" it is in. Typical form modes include, add mode, edit mode, and view mode. If the window is in add mode, it creates a

30   new BO 207 of the appropriate type in this method.

```
Public Sub InitClass(colPrevSelection As CArchCollection)
     If colPrevSelection Is Nothing Then
          ' no accounts were previously selected
```

```
            Set m_colPrevSelection = New CArchCollection
            Set m_colNewSelection = New CArchCollection
        Else
            ' some accounts may have already been selected
            Set m_colPrevSelection = colPrevSelection
            Set m_colNewSelection = colPrevSelection.Clone()
        End If

        Set m_colResults = New CArchCollection

        DetermineFormMode()
    End Sub
```

InitForm

The InitForm procedure of each controller 206 coordinates any initialization of the form 204 before it is displayed. Because initialization is often a multi-step process, InitForm creates the window and then delegates the majority of the initialization logic to helper methods that each have a single purpose, in order to follow the rules of good functional decomposition. For example, the logic to determine a form's 204 state based on user actions and relevant security restrictions and move to that state is encapsulated in the DetermineFormState method.

```
Public Sub InitForm()
    ' create my form
    Set m_frmCurrentForm = New frmAccountSearch
    ' figure out the state of my form based on arguments I
received in InitClass and
    ' enable/disable the appropriate controls
    DetermineFormState()
    ' fill my form with data
    PopulateForm()
End Sub
```

PopulateForm

PopulateForm is a private method responsible for filling the form with data during initialization. It is called exactly once by the InitForm method. PopulateForm is used to fill combo boxes on a form 204, get the details of an object for an editing window, or display objects that have already been selected by the user, as in the following example.

```
Private Sub PopulateForm()
    Dim acct As CAccount
    Dim item As GTListItem

    ' display any accounts already selected by the user
    ' create and add a ListItem for every Account in the
previous selection collection
    With frmCurrentForm.lvwResults.ListItems
        .Clear
        For Each acct In m_colPrevSelection
            Set item = .Add(, acct.Number, acct.Number)
            item.SubItems(1) = acct.Name
        Next
    End With
End Sub
```

ShowForm

The ShowForm method simply centers and displays the newly initialized form 204.

```
Public Sub ShowForm()
    ' center my form
    frmCurrentForm.Move(Screen.Width - frmCurrentForm.Width) /
2,_
                        (Screen.Height - frmCurrentForm.Height)
/ 2
    ' display my form
    frmCurrentForm.Show vbModal  .
End Sub
```

Window Control State Management 306

It is often necessary to enable or disable controls on a form 204 in response to user actions. This section describes the patterns employed by the Component Based Architecture for MTS (CBAM) to manage this process effectively.

Form Mode

It is helpful to distinguish between form mode and form state. Form mode indicates the reason the form 204 has been invoked. Often, forms 204 are used for more than one purpose. A common example is the use of the same form to view, add, and edit a particular type of object, such as a task or a claim. In this case, the form's modes would include View, Add, and Update.

The modes of a form 204 are also used to comply with security restrictions based on the current user's access level. For example, Task Library is a window that limits access to task templates based on the current user's role. It might have a

5   Librarian mode and a Non-Librarian mode to reflect the fact that a non-librarian user cannot be allowed to edit task templates. In this way, modes help to enforce the requirement that certain controls on the form 204 remain disabled unless the user has a certain access level.

10   It is not always necessary for a form 204 to have a mode; a form might be so simple that it would have only one mode – the default mode. In this case, even though it is not immediately necessary, it may be beneficial to make the form "mode-aware" so that it can be easily extended should the need arise.

Form State

15   A form 204 will have a number of different states for each mode, where a state is a unique combination of enabled/disabled, visible/invisible controls. When a form 204 moves to a different state, at least one control is enabled or disabled or modified in some way.

20   A key difference between form mode and form state is that mode is determined when the controller 206 is initialized and remains constant until the controller 206 terminates. State is determined when the window initializes, but is constantly being reevaluated in response to user actions.

Handling UI Events

25   When the value of a control on the form 204 changes, it is necessary to reevaluate the state of the controls on the form (whether or not they are enabled/disabled or visible/invisible, etc.). If changing the value of one control could cause the state of a second control to change, an event handler is written for the appropriate event of the first control.

The following table lists common controls and the events that are triggered when their value changes.

Control	Event
TextBox	Change
ComboBox	Change
ListBox	Click
CheckBox	Click
Option Button	Click

5

The event handler calls the DetermineFormState method on the controller 206.

Setting the State of Controls

It is essential for maintainability that the process of setting the state of controls be separate from the process for setting the values of those controls. The

10 DetermineFormState method on the controller 206 forces this separation between setting the state of controls and setting their values.

DetermineFormState is the only method that modifies the state of any of the controls on the form 204. Because control state requirements are so complex and

15 vary so widely, this is the only restriction made by the architecture framework.

If necessary, parameters are passed to the DetermineFormState function to act as "hints" or "clues" for determining the new state of the form 204. For complex forms, it is helpful to decompose the DetermineFormState function into a number

20 of helper functions, each handling a group of related controls on the form or moving the form 204 to a different state.

Example

The Edit/Add/View Task Window has three modes: Edit, Add, and View. In Add mode, everything on the form is editable. Some details will stay disabled when in Edit mode, since they should be set only once when the task is added. In both Add and Edit modes, the repeat rule may be edited. Enabling editing of the repeat rule always disables the manual editing of the task's due and display dates. In View mode, only the Category combo box and Private checkbox are enabled.

```
' Edit/Add/View Task Form
Private Sub txtName_Change()
    myController.DetermineFormState
End Sub



' Edit/Add/View Task Controller
Public Sub DetermineFormState()
    On Error Goto ErrorHandler
    Select Case m_nFormMode
        ' In Edit Mode, enable only "editable" details and
Repeat Rule editing if necessary
        Case cmFormModeEdit
            EnableAddDetails False
            EnableEditDetails True
            EnableViewDetails True
            If m_frmCurrentForm.chkRepetetiveTask.Checked Then
                EnableEditRepeatRule True
                EnableEditDisplayDueDates False
            Else
                EnableEditRepeatRule False
                EnableEditDisplayDueDates True
            End If
            If m_nFormDirty Then EnableSave True Else
EnableSave False
        ' In Add Mode, enable all details and Repeat Rule
editing if necessary
        Case cmFormModeAdd
            EnableAddDetails True
            EnableEditDetails True
            EnableViewDetails True
            If m_frmCurrentForm.chkRepetetiveTask.Checked Then
                EnableEditRepeatRule True
                EnableEditDisplayDueDates False
            Else
                EnableEditRepeatRule False
                EnableEditDisplayDueDates True
            End If
```

```
                    If m_nFormDirty Then EnableSave True Else
       EnableSave False
                    ' In View Mode, disable everything except a few
       details
 5              Case cmFormModeView
                    EnableAddDetails False
                    EnableEditDetails False
                    EnableViewDetails True
                    EnableEditRepeatRule False
10                  EnableEditDisplayDueDates False
                    EnableSave False
                Case Else
            End Select
            Exit Sub
15      ErrorHandler:
                ' error handling
        End Sub


20      ' Edit/Add/View Task Controller
        Private Sub EnableAddDetails(bYesNo As Boolean)
            On Error Goto ErrorHandler            .
            ' Enable or disable controls that should be available only
        when the task is being added.
25          With frmCurrentForm
                .Name.Enabled = bYesNo
                .Description.Enabled = bYesNo
                .Type.Enabled = bYesNo
                .Level.Enabled = bYesNo
30              .Source.Enabled = bYesNo
            End With
            Exit Sub
        ErrorHandler:
                ' error handling logic
35      End Sub
```

Window Data Validation 308

Window data validation is the process by which data on the window is examined
for errors, inconsistencies, and proper formatting. It is important, for the sake of
consistency, to implement this process similarly or identically in all windows of
the application.

Types of Validation

Input Masking

Input masking is the first line of defense. It involves screening the data (usually character by character) as it is entered, to prevent the user from even entering

5 invalid data. Input masking may be done programmatically or via a special masked text box, however the logic is always located on the form, and is invoked whenever a masked field changes.

Single-Field Range Checking

10 Single-field range checking determines the validity of the value of one field on the form by comparing it with a set of valid values. Single-field range checking may be done via a combo box, spin button, or programmatically on the form, and is invoked whenever the range-checked field changes.

Cross-Field Validation

15 Cross-field validation compares the values of two or more fields to determine if a validation rule is met or broken, and occurs just before saving (or searching). Cross-field validation may be done on the Controller 206 or the Business Object 207, however it is preferable to place the logic on the Business Object 207 when

20 the validation logic can be shared by multiple Controllers 206.

Invalid data is caught and rejected as early as possible during the input process. Input masking and range checking provide the first line of defense, followed by cross-field validation when the window saves (or searches).

Single-Field Validation

25 All single-field validation is accomplished via some sort of input masking. Masks that are attached to textboxes are used to validate the type or format of data being entered. Combo boxes and spin buttons may also be used to limit the user to valid choices. If neither of these are sufficient, a small amount of logic

30 may be placed on the form's event handler to perform the masking functionality,

such as keeping a value below a certain threshold or keeping apostrophes out of a textbox.

Cross-Field Validation

When the user clicks OK or Save, the form calls the IsFormDataValid on the
5 controller to perform cross-field validation (e.g., verifying that a start date is less than an end date). If the business object 207 contains validation rules, the controller 206 may call a method on the business object 207 to make sure those rules are not violated.

10 If invalid data is detected by the controller 206, it will notify the user with a message box and, if possible, the indicate which field or fields are in error. Under no circumstances will the window perform validation when the user is trying to cancel.

Example

15
```
' Generic Edit Form
Private Sub cmdOK_Click()
    On Error Goto ErrorHandler
    ' shut down if my data is valid.
20    ' saving/canceling will occur in my controller's
QueryUnload function
    If IsFormDataValid Then Unload Me
    Exit Sub          '
ErrorHandler:
25    Err.Raise Err.Number                              '
End Sub

Public Function IsFormDataValid() As Boolean
    On Error Goto ErrorHandler
30
    ' assume success
    IsFormDataValid = True

    ' evaluate all validation rules
35    With frmCurrentForm
        ' make sure start date is earlier than end date
        If .txtStartDate.Text > .txtEndDate.Text Then
            IsFormDataValid = False
            MsgBox cmMsgInvalidEndDate
40            .txtEndDate.SetFocus
        ElseIf ...
```

```
                    ` more validation rules
                End If
            End With
```

5
```
            Exit Function
      ErrorHandler:
            ` error handling logic
      End Function
```

10
Window Save Processing 304

Window "Save Processing" involves tracking changes to data on a form 204 and responding to save and cancel events initiated by the user.

Tracking Changes to Form Data

15 Each window within the CBAM application contains a field within its corresponding control object known as the dirty flag. The dirty flag is set to True whenever an end user modifies data within the window. This field is interrogated by the UI Controller 206 to determine when a user should be prompted on Cancel or if a remote procedure should be invoked upon window close.

20
The application shell provides standard processing for each window containing an OK or Save button.

Saving

The default Save processing is implemented within the UI Controller 206 as

25 follows:

The UI Controller is Notified that the OK button has been clicked. Then the controller 206 checks its Dirty Flag. If flag is dirty, the controller 206 calls the InterrogateForm method to retrieve data from the form 204 and calls a server

30 component 222 to store the business object 207 in the database. If the Dirty Flag is not set, then no save is necessary. The window is then closed.

Canceling

When the user cancels a window, the UI Controller 206 immediately examines the Dirty Flag. If the flag is set to true, the user is prompted that their changes will be lost if they decide to close the window.

5

Once prompted, the user can elect to continue to close the window and lose their changes or decide not to close and continue working.

10

Window Shutdown Processing 310

In the event of an error, it is sometimes necessary to shutdown a window or to terminate the entire application. It is critical that all windows follow the shutdown process in order to avoid the GPFs commonly associated with terminating incorrectly. Following is how the window/application is shutdown.

15

Shutdown Scope

The scope of the shutdown is as small as possible. If an error occurs in a controller 206 that does not affect the rest of the application, only that window is

20 shut down. If an error occurs that threatens the entire application, there is a way to quickly close every open window in the application. The window shutdown strategy is able to accommodate both types of shutdowns.

Shutdown

In order to know what windows must be shut down, the architecture tracks which windows are open. Whenever the App Object 202 creates a controller 206, it calls its RegCTLR function to add the controller 206 to a collection of open

5   controllers. Likewise, whenever a window closes, it tells the App Object 202 that it is closing by calling the App Object's 202 UnRegCTLR function, and the App Object 202 removes the closing controller 206 from its collection. In the case of an error, the App Object 202 loops through its collection of open controllers, telling each controller to "quiesce" or shutdown immediately.

10   ### *GeneralErrorHandler*

The GeneralErrorHandler is a method in MArch.bas that acts as the point of entry into the architecture's error handling mechanism. A component or a controller will call the GeneralErrorHandler when they encounter any type of unexpected or unknown error. The general error handler will return a value indicating what the

15   component or controller should do: (1) resume on the line that triggered the error (2) resume on the statement after the line that triggered the error (3) exit the function (4) quiesce (5) shutdown the entire application.

```
     ErrorHandler:
20       Select Case CStr(Err.Number)
             ' handle a search with no result error
           Case cmErrNoClaimTreeData
               MsgBox cmMsgNoResultsQuery, vbInformation
               frmCurrentForm.StatusBar.Panels(1) =
25   cmNoResultsQuery
               'Sets mouse pointer back to default
               frmCurrentForm.MousePointer = vbDefault
           Case Else
               Dim nResumeCode As Integer
30             nResumeCode =
     GeneralErrorHandler(objApp.objArch.AsMsgStruct, cmController,
     —
                                                  cmClassName,
     cmMethodName)
35             Select Case CStr(nResumeCode)
                   Case cmErrorResume
                       Resume
                   Case cmErrorResumeNext
                       Resume Next
```

```
                    Case cmErrorExit
                        Exit Sub
                    Case cmErrorQuiesce
                        Quiesce
                    Case Else
                        objApp.Shutdown
                End Select
        End Select
    End Sub
```

In order to prevent recursive calls the GeneralErrorHandler keeps a collection of controllers that are in the process of shutting down. If it is called twice in a row by the same controller 206, it is able to detect and short-circuit the loop. When the controller 206 finally does terminate, it calls the UnRegisterError function to let the GeneralErrorHandler know that it has shut down and removed from the collection of controllers.

Shutdown Process

After being told what to do by the GeneralErrorHandler, the controller 206 in error may try to execute the statement that caused the error, proceed as if nothing happened, exit the current function, call its Quiesce function to shut itself down, or call the Shutdown method on the App Object 202 to shut the entire application down.

Additional Standard Methods

Searching

Controllers 206 that manage search windows have a public method named Find<Noun>s where <Noun> is the type of object being searched for. This method is called in the event handler for the Find Now button.

Saving

Any controller 206 that manages an edit window has a public method called Save that saves changes the user makes to the data on the form 204. This method is called by the event handlers for both the Save and OK buttons (when/if the OK

5   button needs to save changes before closing).

Closing

A VB window is closed by the user in several ways: via the control-box in upper left corner, the X button in upper right corner, or the Close button. When the form closes, the only method that will always be called, regardless of the way in

10   which the close was initiated, is the form's 204 QueryUnload event handler.

Because of this, there cannot be a standard Close method. Any processing that must occur when a window closes is to be done in the QueryUnload method on the controller 206 (which is called by the form's QueryUnload event handler).

15

The VB statement, Unload Me, appears in the Close button's event handler to manually initiate the unloading process. In this way, the Close button mimics the functionality of the control box and the X button, so that the closing process is handled the same way every time, regardless of how the user triggered the close.

20   The OK button's event handler also executes the Unload Me statement, but calls the Save method on the controller first to save any pending changes.

BUSINESS OBJECTS

25

Business Objects 207 are responsible for containing data, maintaining the integrity of that data, and exposing functions that make the data easy to manipulate. Whenever logic pertains to a single BO 207 it is a candidate to be placed on that BO. This ensures that it will not be coded once for each controller

206 that needs it. Following are some standard examples of business object logic.

Business Logic: Managing Life Cycle State

5 *Overview*

The "state" of a business object 207 is the set of all its attributes. Life cycle state refers only to a single attribute (or a small group of attributes) that determine where the BO 207 is in its life cycle. For example, the life cycle states of a Task are Open, Completed, Cleared, or Error. Business objectives usually involve

10 moving a BO toward its final state (i.e., Completed for a Task, Closed for a Supplement, etc.).

Often, there are restrictions on a BO's movement through its life cycle. For example, a Task may only move to the Error state after first being Completed or

15 Cleared. BOs provide a mechanism to ensure that they do not violate life cycle restrictions when they move from state to state.

Approach

A BO 207 has a method to move to each one of its different life cycle states. Rather than simply exposing a public variable containing the life cycle state of

20 the task, the BO exposes methods, such as Task.Clear(), Task.Complete(), and Task.MarkInError(), that move the task a new state. This approach prevents the task from containing an invalid value for life cycle state, and makes it obvious what the life cycle states of a task are.

Example

```
25      ' CTask Business Object
        Public Sub MarkInError()
            On Error Goto ErrorHandler

            Select Case m_nLifeCycleState
30              ' move to error only if I've already been completed or
        cleared
                Case cmTaskCompleted, cmTaskCleared
```

```
                m_nLifeCycleState = cmTaskInError
            ' otherwise, raise an error
            Case Else
                Err.Raise cmErrInvalidLifeCycleState
        End Select
        Exit Sub
    ErrorHandler:
        Err.Raise Err.Number
    End Sub
```

Business Logic: Operating on Groups of Business Objects

Overview

Sometimes, a BO 207 acts as a container for a group of other BOs. This happens when performing operations involving multiple BOs. For example, to close, a claim ensures that it has no open supplements or tasks. There might be a method on the claim BO - CanClose() - that evaluates the business rules restricting the closing of a claim and return true or false. Another situation might involve retrieving the open tasks for a claim. The claim can loop through its collection of tasks, asking each task if it is open and, if so, adding it to a temporary collection which is returned to the caller.

Example

```
    ' Claim Business Object
    ' Error handling omitted for clarity
    Public Function CanClose() As Boolean
        CanClose = HasOpenTasks() And HasOpenSupplements()
    End Function

    Public Function HasOpenTasks() As Boolean
        ' assume that I have open tasks
        HasOpenTasks = True
        ' loop through all my tasks and exit if I find one that is
    open
        Dim task As CTask
        For Each task In m_colTasks
            If task.IsOpen() Then Exit Function
        Next task
        ' I must not have any open tasks
        HasOpenTasks = False
    End Function

    Public Function HasOpenSupplements() As Boolean
```

```
            ` assume that I have open supplements
            HasOpenSupplements = True
            ` loop through all my supplements and exit if I find one
        that is open
5           Dim supp As CSupplement
            For Each supp In m_colSupplements
                If supp.IsOpen() Then Exit Function
            Next supp
            HasOpenSupplements = False
10      End Function

        Public Function GetOpenTasks() As Collection
            Dim task As CTask
            Dim colOpenTasks As Collection
15          For Each task In m_colTasks
                If task.IsOpen() Then colOpenTasks.Add task, task.Id
            Next task
            Set GetOpenTasks = colOpenTasks
        End Function
20
```

Business Object Structures

Overview

When a BO 207 is added or updated, it sends all of its attributes down to a server

25   component 222 to write to the database. Instead of explicitly referring to each

attribute in the parameter list of the functions on the CCA 208 and server

component 222 , all the attributes are sent in a single variant array. This array is

also known as a structure.

Approach

30   Each editable BO 207 has a method named AsStruct that takes the object's

member variables and puts them in a variant array. The CCA 208 calls this

method on a BO 207 before it sends the BO 207 down to the server component

222 to be added or updated. The reason that this is necessary is that, although

object references can be passed by value over the network, the objects themselves

35   cannot. Only basic data types like Integer and String can be sent by value to a

server component 222 . A VB enumeration is used to name the slots of the

structure, so that the server component 222 can use a symbolic name to access

elements in the array instead of an index. Note that this is generally used only when performing adds or full updates on a business object 207.

In a few cases, there is a reason to re-instantiate the BO 207 on the server side.
The FromStruct method does exactly the opposite of the AsStruct method and initializes the BO 207 from a variant array. The size of the structure passed as a parameter to FromStruct is checked to increase the certainty that it is a valid structure.

When a BO 207 contains a reference to another BO 207, the AsStruct method stores the primary key of the referenced BO 207. For example, the Task structure contains a PerformerId, not the performer BO 207 that is referenced by the task. When the FromStruct method encounters the PerformerId in the task structure, it instantiates a new performer BO and fills in the ID, leaving the rest of the performer BO empty.

Example
```
' CTask Business Object

' enumeration of all task attributes
Public Enum TaskAttributes
    cmTaskId
    cmTaskName
    :
    cmTaskDescription
End Enum

' all task attributes declarations here

' all setter and getter functions here

Public Function AsStruct() As CTask
    On Error Goto ErrorHandler

    ' create and fill structure
    Dim vStruct(cmTaskNumOfAttributes - 1) As Variant
    vStruct(cmTaskId) = m_vId
    vStruct(cmTaskName) = m_sName
    vStruct(cmTaskPerformerId) = m_vPerformerId
    :
    vStruct(cmTaskDescription) = m_sDescription
```

```
        AsStruct = vStruct
        Exit Function
    ErrorHandler:
        Err.Raise Err.Number
5   End Function

    Public Sub FromStruct(vStruct As Variant)
        On Error Goto ErrorHandler

10      ' check size of vStruct
        If Ubound(vStruct) <> (cmTaskNumOfAttributes - 1) Then
    Err.Raise cmErrInvalidParameters

        ' update my values from the structure
15      m_vId = vStruct(cmTaskId)
        m_sName = vStruct(cmTaskName)
        m_vPerformer.Id = vStruct(cmTaskPerformerId)
        :
        m_sDescription = vStruct(cmTaskDescription)
20      Exit Sub
    ErrorHandler:
        Err.Raise Err.Number
    End Sub
```

25

Cloning Business Objects

Overview

Often a copy of a business object 207 is made. Cloning is a way to implement this kind of functionality by encapsulating the copying process in the BO 207

30   itself. Controllers 206 that need to make tentative changes to a business object 207 simply ask the original BO 207 for a clone and make changes to the clone. If the user decides to save the changes, the controller 206 ask the original BO to update itself from the changes made to the clone.

Approach

35   Each BO 207 has a Clone method to return a shallow copy of itself. A shallow copy is a copy that doesn't include copies of the other objects that the BO 207 refers to, but only a copy of a reference to those objects. For example, to clone a task, it does not give the clone a brand new claim object; it gives the clone a new reference to the existing claim object. Collections are the only exception to this

rule – they are always copied completely since they contain references to other BOs.

Each BO 207 also has an UpdateFromClone method to allow it "merge" a clone back in to itself by changing its attributes to match the changes made to the clone.

Example

```
' CTask Business Object
Public Function Clone() As CTask
    On Error Goto ErrorHandler

    ' create clone object
    Dim tskClone As CTask
    Set tskClone = New CTask

    ' fill clone with my data
    With tskClone
        .Id = m_vId
        .Name = m_sName
        .PerformerId = m_vPerformerId
        Set .Performer = m_prfPerformer
        :
        .Description = m_sDescription
    End With

    Set Clone = tskClone
    Exit Function
ErrorHandler:
    Err.Raise Err.Number
End Function

Public Sub UpdateFromClone(tskClone As CTask)
    On Error Goto ErrorHandler

    ' set my values equal to the clone's values
    With tskClone
        m_vId = .ID
        m_sName = .Name
        m_vPerformerId = .PerformerId
        Set m_prfPerformer = .Performer
        :
        m_sDescription = .Description
    End With

    Exit Sub
ErrorHandler:
    Err.Raise Err.Number
End Sub
```

Half-Baked Business Objects

5 *Overview*

BOs 207 occasionally are filled only half-full for performance reasons. This is
done for queries involving multiple tables that return large data sets. Using half-
baked BOs 207 can be an error prone process, so it is essential that the half-
baking of BOs are carefully managed and contained.

10

In most applications, there are two kinds of windows – search windows and
edit/detail windows. Search windows are the only windows that half-bake BOs
207. Generally, half-baking only is a problem when a detail window expecting a
fully-baked BO receives a half-baked BO from a search window.

15 *Approach*

Detail windows refresh the BOs 207 they are passed by the search windows,
regardless of whether or not they were already fully-baked. This addresses the
problems associated with passing half-baked BOs and also helps ensure that the
BO 207 is up-to-date.

20

This approach requires another type of method (besides Get, Add, Update, and
Delete) on the CCA 208: a Refresh method. This method is very similar to a Get
method (in fact, it calls the same method on the server component) but is unique
because it refreshes the data in objects that are already created. The detail

25 window's controller 206 calls the appropriate CCA 208 passing the BO 207 to be
refreshed, and may assume that, when control returns from the CCA 208, the BO
207 will be up-to-date and fully-baked.

This is may not be necessary if two windows are very closely related. If the first

30 window is the only window that ever opens the second, it is necessary for the

second window to refresh the BO 207 passed by the first window if it knows that the BO 207 is baked fully enough to be used.

5

CCAs

CCAs 208 are responsible for transforming data from row and columns in a
10   recordset to business objects 207, and for executing calls to server components 222 on behalf of controllers 206.

Retrieving Business Objects

Overview

15   After asking a component to retrieve data, the CCA 208 marshals the data returned by the component into business objects 207 that are used by the UI Controller 206.

Approach

The marshaling process is as follows:

20

CCAs 208 call GetRows on the recordset to get a copy of its data in a variant array in order to release the recordset as soon as possible. A method exist to coordinate the marshaling of each recordset returned by the component.

25   Only one recordset is coordinated in the marshaling process of a single method. A method exist to build a BO from a single row of a recordset. This method is called once for each row in the recordset by the marshaling coordination method.

Example

30
```
' Task CCA
Public Function GetAllTasks() As Collection
```

```
            On Error Goto ErrorHandler

            ' call a helper method to retrieve tasks
            Dim vRows As Variant
 5          vRows = RetrieveAllTasks

            Dim i As Integer
            Dim task As CTask
            Dim colTasks As Collection
10          Set colTasks = New Collection

            ' vRows is dimmed as column, row.  Loop til I run out of
       rows.
            For i = 0 To Ubound(vRows, 2)
15              ' build BO using helper method
                Set task = BuildTaskFromRow(vRows, i)
                ' add to collection with ID as the key
                colTasks.Add task, task.Id
            Next i
20
            Set MarshalTasks = colTasks
            Exit Function
       ErrorHandler:
            Err.Raise Err.Number
25     End Function

       Private Function RetrieveAllTasks() As Variant
            On Error Goto ErrorHandler

30          ' call my component and get a recordset full of all tasks
            Dim rs As ADOR.Recordset
            Set rs = tskComp.GetAllTasks()

            ' get data in variant array from the recordset
35          GetAllTasks = rs.GetRows

            ' release the recordset ASAP
            rs.Close
            Set rs = Nothing
40
            Exit Function
       ErrorHandler:
            Err.Raise Err.Number
       End Function
45
       Private Function BuildTaskFromRow(vRows As Variant,
       nCurrentRow As Integer, _
                                    Optional task As CTask) As
       CTask
50          On Error Goto ErrorHandler

            ' create task if it wasn't passed
            If task Is Nothing Then Set task = New CTask
```

```
          ' fill task with data
          With task
              .Id = vRows(0, nCurrentRow)
 5            .Name = vRows(1, nCurrentRow)
              .PerformerId = vRows(2, nCurrentRow)
              :
              .Description = vRows(32, nCurrentRow)
          End With
10
          Set BuildTaskFromRow = task
          Exit Function
      ErrorHandler:
          Err.Raise Err.Number
15    End Function
```

Refreshing Business Objects

Overview

20 The logic to refresh BOs 207 is very similar to the logic to create them in the first place. A "refresh" method is very similar to a "get" method, but must use BOs 207 that already exist when carrying out the marshalling process.

Example

```
          ' Task CCA
25    Public Sub RefreshTask(task As CTask)
          On Error Goto ErrorHandler

          ' call a helper method to retrieve tasks
          Dim vRow As Variant
30        vRow = RetrieveTaskWithId(task.Id)

          BuildTaskFromRow vRow, i, task
          Exit Sub
      ErrorHandler:
35        Err.Raise Err.Number
      End Sub

      Private Function RetrieveTaskWithId(vId As Variant) As Variant
          On Error Goto ErrorHandler
40
          ' call my component and get a recordset full of all tasks
          Dim rs As ADOR.Recordset
          Set rs = tskComp.GetTaskWithId(vId)

45        ' get data in variant array from the recordset
          RetrieveTaskWithId = rs.GetRows
```

```
         ' release the recordset ASAP
         rs.Close
         Set rs = Nothing

5        Exit Function
   ErrorHandler:
         Err.Raise Err.Number
   End Function
```

10

Adding Business Objects

Overview

Controllers 206 are responsible for creating and populating new BOs 207. To

15   add a BO 207 to the database, the controller 206 must call the CCA 208, passing

the business object 207 to be added. The CCA 208 calls the AsStruct method on

the BO 207, and pass the BO structure down to the component to be saved. It

then updates the BO 207 with the ID and timestamp generated by the server.

Note the method on the CCA 208 just updates the BO 207.

20   *Example*

```
   ' Task CCA
   Public Sub AddTask(task As CTask)
         On Error Goto ErrorHandler

25       ' call component to add task passing a task structure
         Dim vIdAndTimestamp As Variant
         vIdAndTimestamp = tskComp.AddTask(task.AsStruct())

         ' update ID and Timestamp on task
30       task.Id = vIdAndTimestamp(0)
         task.TimeStamp = vIdAndTimestamp(1)
         Exit Sub
   ErrorHandler:
         Err.Raise Err.Number
35 End Sub
```

Updating Business Objects

Overview

40   The update process is very similar to the add process. The only difference is that

the server component only returns a timestamp, since the BO already has an ID.

Example

```
' Task CCA
Public Sub UpdateTask(task As CTask)
    On Error Goto ErrorHandler

    ' call component to update task passing a task structure
    Dim lTimeStamp As Long
    lTimeStamp = tskComp.AddTask(task.AsStruct())

    ' update Timestamp on task
    task.TimeStamp = lTimeStamp
    Exit Sub
ErrorHandler:
    Err.Raise Err.Number
End Sub
```

Deleting Business Objects

Deleting Overview

Like the add and the update methods, delete methods take a business object 207 as a parameter and do not have a return value. The delete method does not modify the object 207 it is deleting since that object will soon be discarded.

Example

```
' Task CCA
Public Sub DeleteTask(task As CTask)
    On Error Goto ErrorHandler

    ' call component to update task passing a the ID and
Timestamp
    tskComp.DeleteTask task.Id, task.TimeStamp
    Exit Sub
ErrorHandler:
    Err.Raise Err.Number
End Sub
```

SERVER COMPONENT

Server components 222 have two purposes: enforcing business rules and carrying out data access operations. They are designed to avoid duplicating logic between functions.

Designing for Reuse

Enforcing Encapsulation

Each server component 222 encapsulates a single database table or a set of

closely related database tables. As much as possible, server components 222

5   select or modify data from a single table. A component occasionally selects from

a table that is "owned" or encapsulated by another component in order to use a

join (for efficiency reasons). A server component 222 often collaborates with

other server components to complete a business transaction.

Partioning Logic between Multiple Classes

10   If the component becomes very large, it is split into more than one class. When

this occurs, it is divided into two classes – one for business rules and one for data

access. The business rules class implements the component's interface and

utilizes the data access class to modify data as needed.

15   ### *Example*

```
       Private Function MarkTaskInError(vMsg As Variant, _
                                        vTaskId As Variant, _
                                        lTimestamp As Variant, _
20                                      sReason As String) As Long
            On Error GoTo ErrorHandler
            Const cmMethodName = "MarkTaskInError"

            ' set the SQL statement
25          Dim sSQL As String
            sSQL = cmSQLMarkTaskInError

            ' get a new timestamp
            Dim lNewTimeStamp As Long
30          lNewTimeStamp = GetTimeStamp()

            ' create and fill a collection of arguments to be merged
       with
            ' the SQL by the ExecuteQuery method
35          Dim colArgs As CCollection
            Set colArgs = New CCollection
            With colArgs
                .Add lNewTimeStamp
                .Add cmDBBooleanTrue
40              .Add sReason
```

```
            .Add vTaskId
            .Add lTimestamp
        End With

 5      ' run the SQL and set my return value
        ExecuteQuery vMsg, cmUpdate, sSQL, colArguments:=colArgs
        MarkTaskInError = lNewTimeStamp

        ' tell MTS I'm done
10      GetObjectContext.SetComplete
        Exit Function

ErrorHandler:
        ' do error handling here
15  End Function
```

ERROR HANDLING

General Information

With the exception of "Class_Initialize", "Class_Terminate", and methods called within an error handler, every function or subroutine has a user defined 'On Error GoTo' statement. The first line in each procedure is: On Error GoTo ErrorHandler. A line near the end of the procedure is given a label

5 "ErrorHandler". (Note that because line labels in VB 5.0 have procedure scope, each procedure can have a line labeled "ErrorHandler"). The ErrorHandler label is preceded by a Exit Sub or Exit Function statement to avoid executing the error handling code when there is no error.

10 Errors are handled differently based on the module's level within the application (i.e., user interface modules are responsible for displaying error messages to the user).

All modules take advantage of technical architecture to log messages. Client
15 modules that already have a reference to the architecture call the Log Manager object directly. Because server modules do not usually have a reference to the architecture, they use the LogMessage() global function complied into each server component.

20 Any errors that are raised within a server component 222 are handled by the calling UI controller 206. This ensures that the user is appropriately notified of the error and that business errors are not translated to unhandled fatal errors.

All unexpected errors are handled by a general error handler function at the
25 global Architecture module in order to always gracefully shut-down the application.

Server Component Errors

The error handler for each service module contains a Case statement to check for all anticipated errors. If the error is not a recoverable error, the logic to handle it is first tell MTS about the error by calling GetObjectContext.SetAbort(). Next, the global LogMessage() function is called to log the short description intended

5 for level one support personnel. Then the LogMessage() function is called a second time to log the detailed description of the error for upper level support personnel. Finally, the error is re-raised, so that the calling function will know the operation failed.

10 A default Case condition is coded to handle any unexpected errors. This logs the VB generated error then raises it. A code sample is provided below:

Following is an example of how error handling in the task component is implemented when an attempt is made to reassign a task to a performer that

15 doesn't exist. Executing SQL to reassign a task to a non-existent performer generates a referential integrity violation error, which is trapped in this error handler:

```
'Class Declarations
20      Private Const cmClassName = "CTaskComp"

Public Sub ReassignTask(…)
        On Error GoTo ErrorHandler
        Private Const cmMethodName = "ReassignTask"
25      Private Const cmErrReassignTask = "Could not reassign
task."
            :
        ' logic to reassign a task
            :
30      GetObjectContext.SetComplete
        Exit Sub

ErrorHandler:
        Dim sShortDescr As String
35      sShortDescr = cmErrReassignTask
        ' log short description as warning
        LogMessage vMsg, Err.Number, cmSeverityWarning,
cmClassName, cmMethodName, sShortDescr

40      Dim sLongDescr As String
        Select Case Err.Number
```

```
            Case cmErrRefIntegrityViolation
                GetObjectContext.SetAbort
                sLongDescr = "Referential integrity violation -
        tried " _
  5               & "to reassign task to a non-existant
        performer.  " _
                    & "Association ID: " & sAssnId _
                    & "Association Type: " & sAssnType _
                    & "Old Performer Id: " & sOldPerformerId _
 10               & "New Performer Id: " & sNewPerformerId
                ' log long description as severe
                LogMessage vMsg, Err.Number, cmSeveritySevere,
        cmClassName, cmMethodName, _
                        sLongDescr
 15             Err.Raise Err.Number
                :
            ' more error handling
                :
            Case Else
 20             ' let architecture handle unanticipated error
                Dim nResumeCode As Integer
                nResumeCode = GeneralErrorHandler(vMsg, cmServer,
        cmClassName, cmMethodName)
                Select Case nResumeCode
 25                 Case cmErrorResume
                        Resume
                    Case cmErrorResumeNext
                        Resume Next
                    Case cmErrorExit
 30                     Exit Sub
                    Case Else
                        GetObjectContext.Abort
                        Err.Raise Err.Number
                End Select
 35     End Select
        End Sub
```

CCAs, CCIs, Business Objects, and Forms

40   All CCI's, CCA's, Business Objects, and Forms raise any error that is generated.
     A code sample is provided below:

```
Sub SubName()
     On Error GoTo ErrorHandler
45   <the procedure's code here>
     .........
     Exit Sub
ErrorHandler:
     Err.Raise Err.Number
50   End Sub
```

User Interface Controller Errors

The user interface controllers 206 handle any errors generated and passed up from the lower levels of the application. UI modules are responsible for handling whatever errors might be raised by server components 222 by displaying a message box to the user.

Any error generated in the UI's is also displayed to the user in a dialog box. Any error initiated on the client is logged using the LogMessage() procedure. Errors initiated on the server will already have been logged and therefore do not need to be logged again.

All unexpected errors are trapped by a general error method at the global architecture module. Depending on the value returned from this function, the controller may resume on the statement that triggered the error, resume on the next statement, call its Quiesce function to shut itself down, or call a Shutdown method on the application object to shutdown the entire application.

No errors are raised from this level of the application, since controllers handle all errors. A code sample of a controller error handler is provided below:

```
'Class Constants
Private Const cmClassName As String = "<ComponentName>"

Sub SubName()
    On Error GoTo ErrorHandler
    Const cmMethodName As String = "<MethodName>"
    :
    :
    <the procedure's code here>
    :
    :
    Exit Sub
```

```
ErrorHandler:

        Select Case CStr(Err.Number)
            Case …
                'display the error to the user
                ' perform any necessary logic
                Exit Sub (or Resume, or Resume Next)
                :
                :
            Case Else
                Dim nResumeCode As Integer
                nResumeCode = GeneralErrorHandler(vMsg,
        cmController, cmClassName, cmMethodName)
                Select Case CStr(nResumeCode)
                    Case cmErrorResume
                        Resume
                    Case cmErrorResumeNext
                        Resume Next
                    Case cmErrorExit
                        Exit Sub
                    Case cmErrorQuiesce
                        Quiesce
                    Case Else
                        objApp.SHUTDOWN
                End Select
        End Select
End Sub
```

LOCALIZATION

The CBAM application is constructed so that it can be localized for different languages and countries with a minimum effort or conversion.

Requirements and Scope

The CBAM architecture provides support for certain localization features:

Localizable Resource Repository;

Flexible User Interface Design;

Date Format Localization; and

Exposure of Windows Operation System Localization Features.

Localization Approach Checklist

Localization Feature	Supported via Architecture Service	Supported via Architecture API's	Best Practices and Assumptions*
Language Code (Locale Identifier)	✓		
Time Zones			✓
Date/Time		✓	
Name			✓
Telephone Numbers			✓
Functions to Avoid			✓
Weights and Measures			✓
Money			✓
Addresses/Address Hierarchies			✓
Menus, Icons, Labels/Identifiers on Windows			✓
Messages/ Dialogs	✓		
String Functions, Sort Order and String Comparison	✓		
Code Tables	✓		
Drop-Down Lists	✓		
Form & Correspondence Templates			✓
Online and Printed Documentation			✓
Database (DB2)			✓
3rd Party Controls			✓
Miscellaneous			✓

5

Localizable Literals Repository

The CBAM application has an infrastructure to support multiple languages. The architecture acts as a centralized literals repository via its Codes Table Approach.

10

The Codes Tables have localization in mind. Each row in the codes table contains an associated language identifier. Via the language identifier, any given code can support values of any language.

Flexible Interface 400

Flexible user interface 400 and code makes customization easy. The Figure 4
5   illustrates how different languages are repainted and recompiled. For example,
both a English UI 404, and a French UI 406 are easily accommodated. This
entails minimal effort because both UIs share the same core code base 402.
Updates to the UIs are merely be a superficial change.

10   Generic graphics are used and overcrowding is avoided to create a user interface
which is easy to localize.

Data Localization

Language localization settings affect the way dates are displayed on UI's (user
interfaces). The default system display format is different for different
Language/Countries. For Example:

20   English (United States) displays "mm/dd/yy" (e.g., "05/16/98")
English (United Kingdom) displays "dd/mm/yy" (e.g., "16/05/98").

The present inventions UI's employ a number of third-party date controls
including *Sheridan Calendar Widgets* (from Sheridan Software) which allow
25   developers to set predefined input masks for dates (via the controls' Property
Pages; the property in this case is "Mask").

Although the Mask property can be manipulated, the default setting is preferably
accepted (the default setting for Mask is "0 – System Default"; it is set at design
30   time). Accepting the default system settings eliminates the need to code for
multiple locales (with some possible exceptions), does not interfere with intrinsic

Visual Basic functions such as DateAdd, and allows dates to be formatted as strings for use in SQL.

The test program illustrated below shows how a date using the English (United

5   Kingdom) default system date format is reformatted to a user-defined format (in this case, a string constant for use with DB2 SQL statements):

```
Const cmDB2DateAndTime = "mm-dd-yyyy-h.mm.ss"
```

10 **Private Sub cmdConvToDB2_Click()**
```
        Dim sDB2Date As String
        sDB2Date = Format$(SSDateCombo1.Date,
         cmDB2DateAndTime)
        txtDB2String.Text = sDB2Date
```
15 End Sub

Leverage Windows Operation System

The CBAM architecture exposes interface methods on the RegistryService object

20 to access locale specific values which are set from the control panel.

The architecture exposes an API from the RegistryService object which allows access to all of the information available in the control panel. Shown below is the signature of the API:

25
```
        GetRegionalInfo(Info As RegionalInfo) As String
            Where RegionalInfo can be any of the values in the table below:
```

RegionalInfo Values			
CmLanguageId	CmDTDateSeparator	cmDayLongNameMonday	cmMonthLongNameJan
CmLanguageLocalized	CmDTTimeSeparator	cmDayLongNameTuesday	cmMonthLongNameFeb
CmLanguageEnglish	CmDTShortDateFormat	cmDayLongNameWednesday	cmMonthLongNameMar
CmLanguageAbbr	CmLDTongDateFormat	cmDayLongNameThursday	cmMonthLongNameApr
CmLanguageNative	CmDTTimeFormat	cmDayLongNameFriday	cmMonthLongNameMay
CmCountryId	CmDTDateFormatOrdering	cmDayLongNameSaturday	cmMonthLongNameJun
CmCountryLocalized	CmDTLongDateOrdering	cmDayLongNameSunday	cmMonthLongNameJul
CmCountryEnglish	CmDTTimeFormatSpecifier	cmDayAbbrNameMonday	cmMonthLongNameAug
CmCountryAbbr	CmDTCenturyFormatSpecifier	cmDayAbbrNameTuesday	cmMonthLongNameSep
CmCountryNative	CmDTTimeWithLeadingZeros	cmDayAbbrNameWednesday	cmMonthLongNameOct
CmLanguageDefaultId	CmDTDayWithLeadingZeros	cmDayAbbrNameThursday	cmMonthLongNameNov
CmCountryDefaultId	CmDTMonthWithLeadingZeros	cmDayAbbrNameFriday	cmMonthLongNameDec
	CmDTDesignatorAM	cmDayAbbrNameSaturday	cmMonthAbbrNameJan
	CmDTDesignatorPM	cmDayAbbrNameSunday	cmMonthAbbrNameFeb

			cmMonthAbbrNameMar
			cmMonthAbbrNameApr
			cmMonthAbbrNameMay
			cmMonthAbbrNameJun
			cmMonthAbbrNameJul
			cmMonthAbbrNameAug
			cmMonthAbbrNameSep
			cmMonthAbbrNameOct
			cmMonthAbbrNameNov
			cmMonthAbbrNameDec

Get RegionalInfo Example:

```
Private Sub Command1_Click()
    MsgBox "This is the language id for English: " & _
    GetRegionalInfo(cmLanguageId)
End Sub
```

Logical Unit of Work

The Logical Unit of Work (LUW) pattern enables separation of concern between UI Controllers 206 and business logic.

Overview

Normally, when a user opens a window, makes changes, and clicks OK or Save, a server component 222 is called to execute a transaction that will save the user's changes to the database. Because of this, it can be said that the window defines the boundary of the transaction, since the transaction is committed when the window closes.

The LUW pattern is useful when database transactions span windows. For example, a user begins editing data on one window and then, without saving, opens another window and begins editing data on that window, the save process involves multiple windows. Neither window controller 206 can manage the saving process, since data from both windows must be saved as an part of an indivisible unit of work. Instead, a LUW object is introduced to manage the saving process.

The LUW acts as a sort of "shopping bag". When a controller 206 modifies a business object 207, it puts it in the bag to be paid for (saved) later. It might give the bag to another controller 206 to finish the shopping (modify more objects),

5   and then to a third controller who pays (asks the LUW to initiate the save).

Approach

Controllers 206 may have different levels of LUW "awareness":

Requires New: always creates a new LUW;

10   Requires: requires an LUW, and creates a new LUW only if one is not passed by the calling controller;

Requires Existing: requires an LUW, but does not create a new LUW if one is not passed by the calling controller. Raises an error if no LUW is passed; and

Not Supported: is not capable of using an LUW.

15

Controllers 206 that always require a new LUW create that LUW in their ArchInitClass function during initialization. They may choose whether or not to involve other windows in their LUW. If it is desirable for another window to be involved in an existing LUW, the controller 206 that owns the LUW passes a

20   reference to that LUW when it calls the App Object 202 to open the second window. Controllers 206 that require an LUW or require an existing LUW accept the LUW as a parameter in the ArchInitClass function.

LUWs contain all the necessary logic to persist their "contents" – the modified

25   BOs 207. They handle calling methods on the CCA 208 and updating the BOs 207 with new IDs and/or timestamps.

ARCHITECTURE API HIERARCHY

30

Following is an overview of the architecture object model, including a description of each method and the parameters it accepts. Additional sections address the concepts behind specific areas (code caching, message logging, and data access) in more detail.

5

Arch Object

Figure 5 depicts the current properties on the Arch Object 200.

The following are APIs located on the Arch Object 200 which return either a
10 retrieved or created instance of an object which implements the following
interfaces:

CodesMan() 500;

TextMan() 502;

IdMan() 504;

15 RegMan() 506;

LogMan() 508;

ErrMan() 510;

UserMan() 512; and

SecurityMan() 514.

20

AsMsgStruct()

This method on the Arch Object returns a variant structure to pass along a remote message.

25

Syntax:

```
Public Function AsMsgStruct() As Variant
End Function
```

30 *Example:*

```
Dim vMsg As Variant
vMsg =  objArch.AsMsgStruct
```

CodesMan

The following are APIs located on the interface of the Arch Object 200 named

5   CodesMan 500:

CheckCacheFreshness();

FillControl(ctlControl, nCategory, nFillType, [nCodeStatus],

[colAssignedCodes]);

10   FilterCodes(colAllCodes, nCodeStatus);

GetCategoryCodes(nCategory);

GetCodeObject(nCategory, sCode);

GetResourceString(lStringId);

GetServerDate();   .

15   RefreshCache();

RemoveValidDates(sCode, colPassedInAssignedCodes); and

SetServerDate(dtServerDate).

CheckCacheFreshness()

Checks whether the cache has expired, if so refresh.

20

Syntax:

```
Private Sub CheckCacheFreshness()
End Sub
```

25   *Example:*

```
CheckCacheFreshness
```

FillControl()

This API is used to fill listboxes or comboboxes with values from a list of

CodeDecodes. Returns a collection for subsequent lookups to Code objects used

30   to fill controls.

Syntax:

```
Public Function FillControl(ctlControl As Object, nCategory As CodeDecodeCats,
nFillType As CodeDecodeLengths, Optional nCodeStatus As CodeDecodeFilters =
cmValidCodes, Optional colAssignedCodes As CCollection) As CCollection
End Function
```

5

10

Parameters:

ctlControl: A reference to a passed in listbox or combobox.
nCategory: The integer based constant which classified these CodeDecodes from
others. Several of the valid constants include:

15   cmCatTaskType = 1
               cmCatSource
               cmCatTaskStatus
nFillType: The attribute of the CodeDecode which you want to fill. Several of
the valid values include:

20   cmCode
               cmShortDecode
               cmLongDecode
nCodeStatus: Optional value which filters the Code Decodes according to their
Effective and Expiration dates. Several of the valid constants include:

25   cmAllCodes   Pending + Valid + Expired Codes
               cmPendingCodes   Codes whose effective date is greater than the
                                                    current date
               cmValidCodes   Not Pending or Expired Codes
colAssignedCodes: Used when filling a control which should fill and include

30   assigned values.

Example:

```
'Declare an instance variable for States collection on object
Private colStates As CCollection
```

35

```
'Call FillControll API, and set local collection inst var to collection of
codes which were used to fill the controll.  This collection will be used for
subsequent lookups.
Set colStates = objArch.CodesMan.FillControl(frmCurrentForm.cboStates,
```
40   `cmCatStates, cmLongDecode)`

FilterCodes()

Returns a collection of code/decodes that are filtered using their effective and expiration dates based on which nCodeStatus is passed from the fillcontrol method.

45

Syntax:

```
Private Function FilterCodes(colAllCodes As CCollection, nCodeStatus As
CodeDecodeFilters) As CCollection
End Function
```

5

Parameters:

```
colAllCodes:
nCodeStatus:
```

10   *Example:*

```
Set colFilteredCodes = FilterCodes(colCodes, nCodeStatus)
```

GetCategoryCodes()

Returns a collection of CCode objects given a valid category

15   *Syntax:*

```
Public Function GetCategoryCodes(nCategory As CodeDecodeCats) As CCollection
End Function
```

Parameters:

20   nCategory: The integer based constant which classified these CodeDecodes from others.

Example:

```
Dim colMyStates As CCollection
25       Set colMyStates = objArch.CodesMan.GetCategoryCodes(cmCatStates)

         'Below shows an example of looking up the Code value for the currently
         selected state.
         With frmCurrentForm.cboStates
30       If .ListIndex > -1 Then
         Dim objCode As CCode
             Set objCode =  colStates(.ItemData(.ListIndex))
             sStateCode = objCode.Code
         End If
35       End With
```

GetCodeObject()

Returns a valid CCode object given a specific category and code.

Syntax:

40   ```
Public Function GetCodeObject(nCategory As CodeDecodeCats, sCode As String) As
CCode
End Function
```

*Parameters:*

> nCategory:  The integer based constant which classified these CodeDecodes from others.

5

> sCode:  A string indicating the Code attribute of the CodeDecode object.

*Example:*

> frmCurrentForm.lblState = objArch.CodesMan.GetCodeObject(cmCatStates,
> "IL").LongDecode

10 *GetResourceString()*

Returns a string from the resource file given a specific string ID.

*Syntax:*

```
Private Function GetResourceString(lStringId As Long) As String
15      End Function
```

Parameters:

> lStringId: The id associated with the string in the resource file.

20 *Example:*

> sMsg = arch.CodesMan.GetResourceString(CLng(vMessage))

GetServerDate()

Returns the date from the server.

25 *Syntax:*

```
Private Function GetServerDate() As Date
        End Function
```

Example:

30   SetServerDate CCA.GetServerDate

RefreshCache()

Refreshes all of the code obhjects in the cache.

Syntax:

```
35      Private Sub RefreshCache()
        End Sub
```

Example:

```
m_Cache.RefreshCache
```

RemoveValidCodes()

Removes all valid codes from the passed in assigned codes collection, which is used to see which codes are assigned and not valid.

Syntax:

```
Private Sub RemoveValidCodes(sCode As String, colPassedInAssignedCodes As
CCollection)
End Sub
```

Parameters:

```
sCode:  Name of code
colPassedInAssignedCodes:   Codes already in use.
```

Example:

```
RemoveValidCodes codCode.Code, colPassedInAssignedCodes
```

SetServerDate()

Sets the server date.

Syntax:

```
Private Sub SetServerDate(dtServerDate As Date)
End Sub
```

Parameters:

```
dtServerDate:   Date of Server.
```

Example:

```
SetServerDate CCA.GetServerDate
```

TextMan

The following are APIs located on the interface of the Arch Object 200 named TextMan 502.

PairUpAposts();

PairUpAmps(); and

MergeParms ().

PairUpAposts ()

Pairs up apostrophes in the passed string.

.

Syntax:

5

```
Public Function PairUpAposts(sOriginalString As String) As String
End Function
```

Parameters:

sOriginalString: string passed in by the caller

10

Example:

```
Dim sString As String
sString = objArch.TextMan.PairUpAposts("This is Monika's string")
'expected return: sString = "This is Monika''s string"
```

15 ### *PairUpAmps ()*

Pairs up ampersands in the passed string.

Syntax:

```
Public Function PairUpAmps(sOriginalString As String) As String
```
20
```
End Function
```

Parameters:

sOriginalString: string passed in by the caller

25 #### *Example:*

.

```
Dim sString As String
sString = objArch.TextMan.PairUpAmps("Forms&Corr")
 'expected return: sString = "Forms&&Corr"
```

30 ### *MergeParms ()*

Merges string with the passed parameters collection.

Syntax:

```
Public Function MergeParms(sString As String, colParms As CCollection) As
```
35
```
String
End Function
```

Parameters:

```
sOriginalString: string passed in by the caller
colParms As Ccollection: collection of the parameters passed in by the caller
```

5 *Example:*

```
Dim sString As String
sString = objArch.TextMan.MergeParms(sString, colParms)
```

10 **IdMan**

The following are APIs located on the interface of the Arch Object 200 named

IdMan 504:

GetGUID();

15 GetSequenceID();

GetTimeStamp();

GetTrackingNbr(); and

GetUniqueId().

GetGUID ()

20 *Syntax:*

```
Public Function GetGUID()
End Function
```

25 *Example:*

```
Dim vNewGuid As Variant
vNewGuid = objArch.IdMan.GetGUID
```

GetSequenceId ()

30 *Syntax:*

```
Public Function GetSequenceId(sTemplateType As CounterName) As String
End Function
```

Parameters:

35
```
sTemplateType: The string specifying the template requesting a sequence id
               (i.e. cmCountFC = Forms & Corr)
```

Example:

```
frmCurrentForm.txtTemplateNumber = objArch.IdMan.GetSequenceId(cmCountFC)
```

GetTimeStamp ()

Syntax:

```
        Public Function GetTimeStamp()
5       End Function
```

Example:

```
        Dim nNewTimeStamp As Long
        nNewTimeStamp = objArch.IdMan.GetTimeStamp
10
```

GetTrackingNbr ()

15

Syntax:

```
        Public Function GetTrackingNbr()
        End Function
```

20 *Example:*

```
        Set objTechArch = New CTechArch
        sUniqueTrackNum = objTechArch.IdMan.GetTrackingNbr
```

GetUniqueId ()

25 *Syntax:*

```
        Public Function GetUniqueId()
        End Function
```

Example:

```
30      Dim vUid As Variant
        vNewUid = objArch.IdMan.GetUniqueId
```

35   **RegMan**

The following are APIs located on the interface of the Arch Object 200 named
RegMan 506:

GetCacheLife();

40 GetClientDSN();

GetComputerName();

GetDefaultAndValidate();

GetFCArchiveDirectory();

GetFCDistributionDirectory();

5   GetFCMasterDirectory();

GetFCUserDirectory();

GetFCWorkingDirectory();

GetHelpPath();

GetLocalInfo();

10   GetLogLevel();

GetRegionalInfo();

GetRegValue();

GetServerDSN();

GetSetting();

15   GetTimerLogLevel();

GetTimerLogPath(); and

GetUseLocalCodes().

GetCacheLife()

20   *Syntax:*

```
Public Function GetCacheLife() As String
End Function
```

Example:

25
```
Dim s As String
s = objArch.RegMan.GetCacheLife
```

GetClientDSN()

Syntax:

30
```
Public Function GetClientDSN() As String
End Function
```

Example:

```
Dim s As String
```
35
```
s = objArch.RegMan.GetClientDSN
```

GetComputerName()

Syntax:

```
Public Function GetComputerName() As String
End Function
```

Example:

```
Dim s As String
s = objArch.RegMan.GetComputerName
```

GetDefaultAndValidate()

Syntax:

```
Private Function GetDefaultAndValidate(sKey As String) As String
End Function
```

Parameters:

sKey: The key within the registry of which the user is requesting (i.e.: Help
Path)

Example:

```
Dim sDefault As String
sDefault = objArch.RegMan.GetDefaultAndValidate(sKey)
```

GetFCArchiveDirectory()

Syntax:

```
Public Function GetFCArchiveDirectory() As String
End Function
```

Example:

```
Dim s As String
s = objArch.RegMan.GetFCArchiveDirectory
```

GetFCDistributionDirectory()

Syntax:

```
Public Function GetFCDistributionDirectory() As String
End Function
```

Example:

```
Dim s As String
s = objArch.RegMan.GetFCDistributionDirectory
```

GetFCMasterDirectory()

Syntax:

```
Public Function GetFCMasterDirectory() As String
End Function
```

5

10   *Example:*

```
Dim s As String
s = objArch.RegMan.GetFCMasterDirectory
```

GetFCUserDirectory()

15   *Syntax:*

```
Public Function GetFCUserDirectory() As String
End Function
```

Example:

20

```
Dim s As String
s = objArch.RegMan.GetFCUserDirectory
```

GetFCWorkingDirectory()

Syntax:

25

```
Public Function GetFCWorkingDirectory() As String
End Function
```

Example:

```
Dim s As String
s = objArch.RegMan.GetFCWorkingDirectory
```

30

GetHelpPath()

Syntax:

```
Public Function GetHelpPath() As String
End Function
```

35

Example:

```
Dim s As String
s = objArch.RegMan.GetHelpPath
```

GetLocalInfo()

Syntax:

```
Public Function GetLocalInfo() As String
End Function
```

Example:

```
Dim s As String
s = objArch.RegMan.GetLocalInfo
```

GetLogLevel()

Syntax:

```
Public Function GetLogLevel() As String
End Function
```

Example:

```
Dim s As String
s = objArch.RegMan.GetLogLevel
```

GetRegionalInfo()

Allows access to all locale specific values which are set from control panel.

Syntax:

```
Public Function GetRegionalInfo(Info As RegionalInfo) As String   .
End Function
```

Parameters:

```
Info:  string containing the regional information. Several of the valid
constants include:
      • cmLanguageId = &H1                  '  language id
      • cmLanguageLocalized = &H2           '  localized name of language
      • cmLanguageEnglish = &H1001          '  English name of language
      • cmLanguageAbbr = &H3                '  abbreviated language name
      • cmLanguageNative = &H4              '  native name of language
```

Example:

```
Dim s As String
s = objArch.RegMan.GetRegionalInfo
```

GetRegValue()

5   *Syntax:*

```
Public Function GetRegValue() As String
End Function
```

Example:

10   ```
Dim s As String
s = objArch.RegMan.GetRegValue
```

GetServerDSN()

Syntax:

15   ```
Public Function GetServerDSN() As String
End Function   .
```

*Example:*

```
Dim s As String
20     s = objArch.RegMan.GetServerDSN
```

GetSetting()
Get setting from the registry.

Syntax:

25   ```
Public Function GetSetting(sKey As String) As String
End Function
```

Parameters:

sKey: The key within the registry of which the user is requesting (i.e.:
30   Help Path)

Parameters:

```
GetHelpPath = GetSetting(cmRegHelpPathKey)
```

GetTimerLogLevel()

35

Syntax:

```
Public Function GetTimerLogLevel() As String
End Function
```

Example:

```
Dim s As String
s = objArch.RegMan.GetTimerLogLevel
```

5 *GetTimerLogPath()*

Syntax:

```
Public Function GetTimerLogPath() As String
End Function
```

10

Example:

```
Dim s As String
s = objArch.RegMan.GetTimerLogPath
```

GetUseLocalCodes()

15 *Syntax:*

```
Public Function GetUseLocalCodes() As String
End Function
```

Example:

20
```
Dim s As String
s = objArch.RegMan.GetUseLocalCodes
```

LPSTRToVBString()

Extracts a VB string from a buffer containing a null terminated string.

25 *Syntax:*

```
Private Function LPSTRToVBString$(ByVal s$)
End Function
```

30

LogMan

The following are APIs located on the interface of the Arch Object 200 named
LogMan 508:

35 LogMessage ();

WriteToDatabase(); and

WriteToLocalLog().

LogMessage ()

Used to log the message. This function will determine where the message
should be logged, if at all, based on its severity and the vMsg's
log level.

5

Syntax:

```
Public Sub LogMessage(vMsg As Variant, _
                      lSeverity As Long, _
                      sClassName As String, _
                      sMethodName As String, _
                      sVersion As String, _
                      lErrorNum As Long, _
                      Optional sText As String = vbNullString)
End Sub
```

15

Parameters:

```
vMsg: the standard architecture message
lSeverity: the severity of the message
sClassName: the name of the class logging the message
sMethodName: the name of the method logging the message
sVersion: the version of the binary file (EXE or DLL) that contains the method
logging message
lErrorNum: the number of the current error
sText: an optional parameter containing the text of the message.  If omitted,
the text will be looked up in a string file or the generic VB error
description will be used
```

Example:

```
If Err.Number <> 0 Then
    ' log message
    Arch.LogMan.LogMessage(vMsg, cmSeverityFatal, "COrganizationCTLR",
"InitForm",
                              GetVersion(), Err.Number, Err.Description)
    ' re-raise the error
    Err.Raise Err.Number
End If
```

WriteToDatabase ()

Used to log the message to the database on the server using the CLoggingComp.
This function returns the TrackingId that is generated by the CLoggingObject.

40

Syntax:

```
Private Sub WriteToDatabase(vMsg As Variant, msgToLog As CMessage)
End Sub
```

45

Parameters:

```
vMsg: the standard architecture message
msgToLog: a parameter containing the text of the message.
```

5 *Example:*

```
If msgToLog.IsLoggableAtLevel(m_lLocalLogLevel) Then
    WriteToDatabase vMsg, msgToLog
End If
```

WriteToLocalLog ()

10 Used to log the message to either a flat file, in the case of Windows 95, or the NT Event Log, in the case of Windows NT.

Syntax:

```
Private Sub WriteToLocalLog(msgToLog As CMessage)
End Sub
```

15

Parameters:

```
msgToLog: a parameter containing the text of the message.
```

20 *Example:*

```
ErrorHandler:
    WriteToLocalLog msgToLog
End Sub
```

25

ErrMan

The following are APIs located on the interface of the Arch Object 200 named

30 ErrMan 510:

HandleError();

RaiseOriginal();

ResetError(); and

Update().

35 ### *HandleError()*

This method is passed through to the general error handler in MArch.bas

Syntax:

```
Public Function HandleError(vMsg As Variant, nCompType As CompType, sClassName
As String, sMethodName As String) As ErrResumeCodes
End Sub
```

40

Parameters:

```
vMsg:       General Architecture Information
nCompType: Contains tier information (Client or Server)
sClassName:      Class which raised the error.
sMethodName:      Method which raised the error.
```

5

RaiseOriginal()

This method is used to Reset the error object and raise.

10

Syntax:

```
Public Sub RaiseOriginal()
End Sub
```

15 *Example:*

```
objArch.ErrMan.RaiseOriginal
```

ResetError()

This method is used to reset attributes.

20 *Syntax:*

```
Public Sub ResetError()
End Sub
```

Example:

25
```
objArch.ErrMan.ResetError
```

30 ## *Update()*

This method is used to update attributes to the values of VBs global Error object.

Syntax:

```
Public Sub Update()
35  End Sub
```

Example:

```
objArch.ErrMan.Update
```

40

UserMan

The following are APIs located on the interface of the Arch Object 200 named

5   UserMan 512.

UserId;

EmployeeId;

EmployeeName;

EmployeeFirstName;

10   EmployeeLastName;

EmployeeMiddleInitial;

GetAuthorizedEmployees;

IsSuperOf ();

IsRelativeOf(); and

15   IsInRole().

UserId()

Syntax:

```
Public Property Get UserId() As String
End Property
```
20

Example:

```
Dim sNewUserId As String
sNewUserId = objArch.UserMan.UserId
```

25   *EmployeeId()*

Syntax:

```
Public Property Get EmployeeId() As String
End Property
```
30

Example:

```
Dim sNewEmployeeId As String
sNewEmployeeId = objArch.UserMan.EmployeeId
```

EmployeeName()

5   *Syntax:*

```
Public Property Get EmployeeName() As String
End Property
```

Example:

10
```
Dim sName As String
sName = objArch.UserMan.EmployeeName
```

EmployeeFirstName()

Syntax:

15
```
Public Property Get EmployeeFirstName() As String
End Property
```

Example:

```
Dim sFName As String
20      sFName = objArch.UserMan.EmployeeFirstName
```

EmployeeLastName()

Syntax:

```
Public Property Get EmployeeLastName()As String
25      End Property
```

Example:

```
Dim sLName As String
sLName = objArch.UserMan.EmployeeLastName
```

30   ### *EmployeeMiddleInitial()*

Syntax:

```
Public Property Get EmployeeMiddleInitial() As String
End Property
```

35

Example:

```
Dim sMI As String
sMI = objArch.UserMan.EmployeeMiddleInitial
```

GetAuthorizedEmployees()

Creates a collection of user's supervisees from the dictionary and returns GetAuthorizedEmployees - collection of authorized employees

5 *Syntax:*

```
Public Function GetAuthorizedEmployees() As CCollection
End Function
```

Example:

10
```
Dim colAuth As Collection
colAuth = objArch.UserMan.GetAuthorizedEmployees
```

IsSuperOf ()

Checks if the current user is supervisor of the passed in user.

15

Syntax:

```
Public Function IsSuperOf(sEmpId As String) As Boolean
End Function
```

20 *Parameters:*

```
sEmpId: string containing Employee ID number
```

Example:

```
Dim bIsSuperOfMonika As Boolean
```
25
```
bIsSuperOfMonika = objArch.UserMan.IsSuperOf("TS012345")
```

IsRelativeOf ()

Checks if the passed in user is relative of the current user.

Syntax:

30
```
Public Function IsRelativeOf(sEmpId As String) As Boolean
End Function
```

Parameters:

```
sEmpId: string containing Employee ID number
```
35

Example:

```
Dim bIsRelativeOfMonika As Boolean
bIsRelativeOfMonika = objArch.UserMan.IsRelativeOf("TS012345")
```

IsInRole ()

Checks to see if the current user is in a certain role.

Syntax:

```
Public Function IsInRole(sRole As String) As Boolean
End Function
```

Parameters:

```
sRole: string containing role
```

Example:

```
Dim bIsInRoleTaskLibrarian As Boolean
bIsInRoleTaskLibrarian = objArch.UserMan.IsInRole("TA")
```

SecurityMan

The following APIs are located on the interface of the Arch Object 200 named SecurityMan 514.

EvalClaimRules;

EvalFileNoteRules;

EvalFormsCorrRules;

EvalOrgRules;

EvalRunApplicationRules;

EvalRunEventProcRules;

EvalTaskTemplateRules;

EvalUserProfilesRules;

IsOperAuthorized;

GetUserId; and

OverrideUser.

EvalClaimRules ()

This API references business rules for Claim security checking and returns a boolean if rules are met.

5 *Syntax:*

```
Private Function EvalClaimRules(lBasicOp As cmBasicOperations, vContextData As
Variant) As Boolean
End Function
```

10 *Parameters:*

```
lBasicOp:   a basic operation the current user is wishing to perform (i.e.
Delete)
vContextData:   a variant array holding relevant business objects or other
information.
```

15

Example:

```
Select Case lOperation
Case cmWorkOnClaim
20              IsOperAuthorized = EvalClaimRules(cmView, vContextData) And _
                                EvalClaimRules(cmEdit, vContextData)
```

EvalFileNoteRules ()

This API references business rules for FileNote security checking and returns a boolean if rules are met.

25 *Syntax:*

```
Private Function EvalFileNoteRules(lBasicOp As cmBasicOperations, vContextData
As Variant) As Boolean
End Function
```

30 *Parameters:*

```
lBasicOp:   a basic operation the current user is wishing to perform (i.e.
Delete)
vContextData:   a variant array holding relevant business objects or other
35  information.
```

Example:

```
Select Case lOperation
40  Case cmDeleteFileNote
                IsOperAuthorized = EvalFileNoteRules(cmDelete, vContextData)
```

EvalFormsCorrRules ()

This API references business rules for Forms and Corr security checking and returns a boolean if rules are met.

Syntax:

```
Private Function EvalFormsCorrRules(lBasicOp As cmBasicOperations) As Boolean
End Function
```

Parameters:

lBasicOp: a basic operation the current user is wishing to perform (i.e. Delete)

Example:

```
Select Case lOperation
Case cmMaintainFormsCorr
            IsOperAuthorized = EvalFormsCorrRules(cmEdit) And _
                               EvalFormsCorrRules(cmDelete) And _
                               EvalFormsCorrRules(cmAdd)
```

EvalOrgRules ()

This API references business rules for Event Processor security checking and returns a boolean if rules are met.

Syntax:

```
Private Function EvalOrgRules(lBasicOp As cmBasicOperations) As Boolean
End Function
```

Parameters:

lBasicOp: a basic operation the current user is wishing to perform (i.e. Delete)

Example:

```
Select Case lOperation
Case cmMaintainOrg
            IsOperAuthorized = EvalOrgRules(cmAdd) And _
                        ·      EvalOrgRules(cmEdit) And _
                               EvalOrgRules(cmDelete)
```

EvalRunApplicationRules ()

This API references business rules for running the application and returns a boolean if rules are met.

Syntax:

```
Private Function EvalRunApplicationRules(lBasicOp As cmBasicOperations) As
Boolean
End Function
```

Parameters:

```
lBasicOp:  a basic operation the current user is wishing to perform (i.e.
Delete)
```

Example:

```
Select Case lOperation
        Case cmRunApplication
                IsOperAuthorized = EvalRunApplicationRules(cmExecute)
```

EvalRunEventProcRules ()

This API references business rules for Event Processor security checking and returns a boolean if rules are met.

Syntax:

```
Private Function EvalRunEventProcRules(lBasicOp As cmBasicOperations) As
Boolean
End Function
```

Parameters:

```
lBasicOp:  a basic operation the current user is wishing to perform (i.e.
Delete)
```

Example:

```
Select Case lOperation
Case cmRunEventProcessor
                IsOperAuthorized = EvalRunEventProcRules(cmExecute)
```

EvalTaskTemplateRules ()

This API references business rules for Task Template security checking and
returns a boolean if rules are met.

Syntax:

```
Private Function EvalTaskTemplateRules(lBasicOp As cmBasicOperations) As
Boolean
End Function
```

Parameters:

```
lBasicOp:  a basic operation the current user is wishing to perform (i.e.
Delete)
```

Example:

```
Select Case lOperation
Case cmMaintainTaskLibrary
            IsOperAuthorized = EvalTaskTemplateRules(cmAdd) And _
                               EvalTaskTemplateRules(cmEdit) And _
                               EvalTaskTemplateRules(cmDelete)
```

EvalUserProfileRules ()

This API references business rules for Task Template security checking and
returns a boolean if rules are met.

Syntax:

```
Private Function EvalUserProfileRules(lBasicOp As cmBasicOperations,
vContextData As Variant) As Boolean
End Function
```

Parameters:

```
lBasicOp:  a basic operation the current user is wishing to perform (i.e.
Delete)
vContextData:  a variant array holding relevant business objects or other
information.
```

Example:

```
Select Case lOperation
Case cmIsRelativeOf
            IsOperAuthorized = EvalUserProfileRules(cmView, vContextData) And
            _

                               EvalUserProfileRules(cmAdd, vContextData) And _
```

```
                                    EvalUserProfileRules(cmEdit, vContextData) And

                                    EvalUserProfileRules(cmDelete, vContextData)
```

5 *GetUserId ()*

Returns the login name/user id of the current user.

Syntax:

```
     Public Function GetUserId() As String
10   End Function
```

Example:

```
     Dim sUserId as String
     sUserId = GetUserId
```

15 *IsOperAuthorized ()*

This API references business rules and returns a boolean determining whether the user has security privileges to perform a certain operation.

Syntax:

```
20   Public Function IsOperAuthorized(vMsg, as variant, nOperation as cmOperations,
     vContext As   Variant) As Boolean
     End Function
```

Parameters:

```
25   vMsg: the standard architecture message
     nOperation:  an enumeration containing name of operation to be checked.
     vContext:  a variant array holding relevant business objects or other
     information.
```

30

Example:

```
     Dim bCanIDoThis As Boolean
     bCanIDoThis = objArch.SecurityMan.IsOperAuthorized(vMsg,aOperationName,
     vContext)
35   TlbEditIcon.Enabled = bCanIDoThis
```

OverrideUser ()

Re-initializes for a different user.

40

Syntax:

```
     Public Sub OverrideUser(Optional sUserId As String, Optional dictRoles As
     CDictionary, Optional dictSubs As CDictionary)
     End Function
```

Parameters:

```
      sUserId:
      dictRoles:
5     dictSubs:
```

Example:

```
Dim x As New CTechArch
10     x.SecurityMan.OverrideUser "Everyone", New CDictionary, New CDictionary
```

15

CODES FRAMEWORK

General Requirements

20   Separate tables (CodesDecodes) are Created for storing the static values.

Only the references to codes/decodes are stored in business tables (e.g., Task)

which utilize these values. This minimizes the size of the business tables, since

storing a Code value takes much less storage space than its corresponding

25   Decode value (e.g., For State, "AL" is stored in each table row instead of the

string "Alabama").

CodeDecodes are stored locally on the client workstation in a local DBMS. On

Application startup, a procedure to ensure the local tables are in sync with the

30   central DBMS is performed.

Infrastructure Approach

The present invention's Code Decode Infrastructure 600 Approach outlines the method of physically modeling codes tables. The model allows codes to be
5   extended with no impact to the physical data model and/or application and architecture. Figure 6 shows the physical layout of CodeDecode tables according to one embodiment of the present invention.

Infrastructure

10 The physical model of the CodeDecode infrastructure 600 does the following:
Supports relational functionality between CodeDecode objects;
Supports extensibility without modification to the DBMS or Application Architecture;

15 Provides a consistent approach for accessing all CodeDecode elements; and

Is easily maintainable.

These generic tables are able to handle new categories, and modification of
20 relationships without a need to change the DBMS or CodeDecode Application Architecture.

Benefits of this model are extensibility and maintainability. This model allows for the modifications of code categories without any impact to the DBMS or the
25 Application Architecture code. This model also requires fewer tables to maintain. In addition, only one method is necessary to access CodeDecodes.

Table Relationships and Field Descriptions:
   (pk) indicates a Primary Key

30

Code_Category 602

- C_Category (pk): The category number for a group of codes
- C_Cache (currently not utilized): Can indicate whether the category should be cached in memory on the client machine
- T_Category: A text description of the category (e.g., Application Task Types, Claim Status, Days of Week)
- D_Last_Update: The date any data within the given category was last updated; this field is used in determining whether to update a category or categories on the local data base

Relationships

- A one-to-many relationship with the table Code (i.e., one category can have multiple codes)

Code 604

- C_Category (pk): The category number for a group of codes
- C_Code (pk): A brief code identifier (up to ten characters; the current maximum length being used is five characters)
- D_Effective: A date field indicating the code's effective date
- D_Expiration: A date field indicating the code's expiration date (the default is January 1, 2999)

Relationships

- A many-to-one relationship with Code_Category 602 (described above)
- A one-to-many relationship with Code_Relations 606 (a given category-and-code combination can be related to multiple other category-and-code combinations)

Code_Relations 606

- C_Category1 (pk): The first category
- C_Code1 (pk): The first code
- C_Category2 (pk): The related category

- C_Code2 (pk): The related code

Relationships

- A many-to-one relationship with the Code table (each category and code in the Code table can have multiple related category-code combinations)

Code_Decode 608

- C_Category (pk): The category number for a group of codes
- C_Code (pk): A brief code identifier (up to ten characters; the current maximum length being used is five characters)
- N_Lang_ID (pk): A value indicating the local language setting (as defined in a given machine's Regional Settings). For example, the value for English (United States) is stored as 0409. Use of this setting allows for the storage and selection of text code descriptions based on the language chosen
- T_Short_Desc: An abbreviated textual description of C_Code
- T_Long_Desc: A full-length textual description of C_Code—what the user will actually see (e.g., Close Supplement – Recovery, File Note, Workers Compensation)

Localization Support Approach

Enabling Localization

Codes have support for multiple languages. The key to this feature is storing a language identifier along with each CodeDecode value. This Language field makes up a part of the compound key of the Code_Decode table. Each Code API lookup includes a system level call to retrieve the Language system variable. This value is used as part of the call to retrieve the values given the correct language.

Maintaining Language Localization Setting

A link to the Language system environment variable to the language keys is
5   stored on each CodeDecode. This value is modified at any time by the user simply by editing the regional settings User Interface available in the Microsoft Windows Control Panel folder.

10

Codes Expiration Approach

Handling Time Sensitive Codes becomes an issue when filling controls with a list of values. One objective is to only allow the user to view and select appropriate
15   entries. The challenge lies in being able to expire Codes without adversely affecting the application. To achieve this, consideration is given to how each UI will decide which values are appropriate to show to the user given its current mode.

20   The three most common UI modes that affect time sensitive codes are Add Mode, View Mode, and Edit Mode.

Add Mode

In Add Mode, typically only valid codes are displayed to the user as selection
25   options. Note that the constant, cmValidCodes, is the default and will still work the same even when this optional parameter is omitted.

      Set colStates = objArch.CodesMan.FillControl(frmCurrentForm.cboStates, cmCatStates,
30   cmLongDecode, **cmValidCodes)**

View Mode

In View Mode, the user is typically viewing results of historical data without direct ability to edit. Editing selected historical data launches another UI. Given this the controls are filled with valid and expired codes, or in other words, non-

5   pending codes.

```
Set colStates = objArch.CodesMan.FillControl(frmCurrentForm.cboStates,
cmCatStates,
    cmLongDecode, cmNonPendingCodes)
```

10

Edit Mode

In Edit Mode, changes are allowed to valid codes but also expired codes are displayed if already assigned to the entity.

15
```
Dim colAssignedCodes As New cCollection
colAssignedCodes.Add HistoricalAddress.State
Set colStates = objArch.CodesMan.FillControl(frmCurrentForm.cboStates,
cmCatStates,
    cmLongDecode, cmValidCodes, colAssignedCodes)
```

20

Updating Local CodeDecodes

25 The Local CodeDecode tables are kept in sync with central storage of CodeDecodes. The architecture is responsible for making a check to see if there are any new or updated code decodes from the server on a regular basis. The architecture also, upon detection of new or modified CodeDecode categories, returns the associated data, and performs an update to the local database. Figure

30 7 is a logic diagram for this process 700.

After an API call, a check is made to determine if the Arch is initialized 702. If it is a check is made to determine if the Freshness Interval has expired 704. If the Freshness Interval has not expired, the API call is complete 706. However, if either the Arch is not initialized or the Freshness Interval has expired, then the

5 "LastUpdate" fields for each category are read from the CodeDecode and passed to the server 708. Then new and updated catagories are read from the database 710. Finally the Local database is updated 712.

Code Access APIs

10

The following are APIs located on the interface of the Arch Object 200 named CodesMan 500.

GetCodeObject(nCategory, sCode);

GetCategoryCodes(nCategory);

15 FillControl(ctlControl, nCategory, nFillType, [nCodeStatus],

[colAssignedCodes]).

GetCodeObject: Returns a valid CCode object given a specific category and code.

20

Syntax: .

```
GetCodeObject(nCategory, sCode)
```

25 *Parameters:*

nCategory: The integer based constant which classified these CodeDecodes from others.
sCode: A string indicating the Code attribute of the CodeDecode object.

30

Example:

```
frmCurrentForm.lblState = objArch.CodesMan.GetCodeObject (cmCatStates,
"IL").LongDecode
```

35

GetCategoryCodes: Returns a collection of CCode objects given a valid category

Syntax:

40

```
GetCategoryCodes(nCategory)
```

Parameters:

nCategory: The integer based constant which classified these CodeDecodes
5   from others.

Example:

```
Dim colMyStates As CCollection
Set colMyStates = objArch.CodesMan.GetCategory(cmCatStates)
```

FillControl: This API is used to fill listboxes or comboboxes with values from a
list of CodeDecodes. Returns a collection for subsequent lookups to Code
objects used to fill controls.

Syntax:

FillControl(ctlControl, nCategory, nFillType, [nCodeStatus], [colAssignedCodes])

Parameters:

ctlControl: A reference to a passed in listbox or combobox.
nCategory: The integer based constant which classified these CodeDecodes
from others.
nFillType: The attribute of the CodeDecode which you want to fill. Valid
values include:

```
cmCode
cmShortDecode
cmLongDecode
```

nCodeStatus: Optional value which filters the Code Decodes according to
their Effective and Expiration dates. Valid constants include the following:

```
cmAllCodes          Pending + Valid + Expired Codes
cmPendingCodes      Codes whose effective date is greater than the
current date
cmValidCodes        Not Pending or Expired Codes
cmExpiredCodes      Codes whose expired date is greater than the current
date
cmNonPendingCodes   Valid + Expired Codes
cmNonValidCodes     Pending + Expired Codes
cmNonExpiredCodes   Pending + Valid Codes
```

colAssignedCodes: Used when filling a control which should fill and
include assigned values.

Example:

```
'Declare an instance variable for States collection on object
Private colStates As CCollection
```

'Call FillControl API, and set local collection inst var to collection of codes which were used to fill the control. This collection will be used for subsequent lookups.

```
Set colStates = objArch.CodesMan.FillControl(frmCurrentForm.cboStates,
cmCatStates, cmLongDecode)
```

'Below shows an example of looking up the Code value for the currently selected state.

```
With frmCurrentForm.cboStates
    If .ListIndex > -1 Then
        Dim objCode As CCode
        Set objCode =  colStates(.ItemData(.ListIndex))
        sStateCode = objCode.Code
    End If
End With
```

Relational Codes Access APIs

Code objects returned via the "GetCodeObject" or "GetCategoryCodes" APIs can have relations to other code objects. This allows for functionality in which codes are associated to other individual code objects.

The APIs used to retrieve these values are similar to those on the CodesMan interface. The difference, however is that the methods are called on the Codes object rather that the CodesManager interface: Listed below again are the APIs.

GetCodeObject(nCategory, sCode);

GetCategoryCodes(nCategory);

FillControl(ctlControl, nCategory, nFillType, [nCodeStatus], [colAssignedCodes]).

Given below is some sample code to illustrate how these APIs are also called on Code objects.

GetCodeObject Example:

```
Dim objBondCode As CCode
Set objBondCode = objArch.CodesMan.GetCodeObject(cmCatLOB, "B")
Dim objSuretyCode As CCode
Set objSuretyCode = objBondCode.GetCodeObject(cmCatSupplement, "B01")
```

GetCategory Example:

```
      Dim objBondCode As CCode
      Set objBondCode = objArch.CodesMan.GetCodeObject(cmCatLOB, "B")
5     Dim colSupplements As CCollection
      Set colSupplements = objBondCode.GetCategory(cmCatSupplement)
```

FillControl Example:

```
10    Dim objBondCode As CCode
      Set objBondCode = objArch.CodesMan.GetCodeObject(cmCatLOB, "B")
      Dim colSupplements As CCollection
      Set colSupplements = objBondCode.FillControl(frmForm.cboSupplements,
      cmCatSupplements,
15          cmLongDecode)
```

<u>MESSAGE LOGGING</u>

20   The message logging architecture allows message logging in a safe and consistent

manner. The interface to the message logging component is simple and

consistent, allowing message logging on any processing tier. Both error and

informational messages are logged to a centralized repository.

25   Abstracting the message logging approach allows the implementation to change

without breaking existing code.

Best Practices

Messages are always logged by the architecture when an unrecoverable error

30   occurs (i.e., the network goes down) and it is not explicitly handled. Message

logging may be used on an as-needed basis to facilitate the diagnosis and fixing

of SIRs. This sort of logging is especially useful at points of integration between

classes and components. Messages logged for the purpose of debugging have a

severity of Informational, so as not to be confused with legitimate error

35   messages.

40

Usage

A message is logged by calling the LogMessage() function on the architecture.

Description of Parameters:

vMsg: the standard architecture message

5   lSeverity: the severity of the message

sClassName: the name of the class logging the message

sMethodName: the name of the method logging the message

sVersion: the version of the binary file (EXE or DLL) that contains the method

logging the message

10   lErrorNum: the number of the current error

sText: an optional parameter containing the text of the message. If omitted, the

text will be looked up in a string file or the generic VB error description will be

used.

sText: an optional parameter containing the text of the message. If omitted, the

15   text will be looked up in a string file or the generic VB error description will be

used.

lLoggingOptions : an optional parameter containing a constant specifying where

to log the message (i.e., passing cmLogToDBAndEventViewer to LogMessage

will log the error to the database and the event viewer.)

20   ***Logging Levels***

Before a message is logged, its severity is compared to the log level of the current

machine. If the severity of the message is less than or equal to the log level, then

the message is logged.

25   Valid values for the log level are defined as an enumeration in VB. They

include:

Value	Name	Description	Example
0	CmFatal	A critical condition that closes or threatens the entire system	Application Server crash
1	CmSevere	A condition that closes or threatens a	Network failure

		major component of the entire system	
2	CmWarning	A warning that something in the system is wrong but it does not close or threaten to close the system	Optimistic locking error
3	CmInformational	Notification of a particular occurrence for logging and audit purposes	Developer debugging information

Example

```
If Err.Number <> 0 Then
    ' log message
    Arch.LogMan.LogMessage(vMsg, cmSeverityFatal, "COrganizationCTLR",
"InitForm",
                        GetVersion(), Err.Number, Err.Description)
    ' re-raise the error
    Err.Raise Err.Number
End If
```

Database Log

The database log table is composed of the following fields:

Field Name	Description
N_MSG_ID	Unique ID of the message
D_MSG	Date the message occurred
C_ERR_SEV	Severity of the error
N_USER_ID	Name of user when error occurred
N_MACH_ID	Name of the machine that the error occurred on
M_CLASS	Name of the class that the error occurred in
M_METHOD	Name of the method that the error occurred in
N_CMPNT_VER	Version of the binary file that the error occurred in
C_ERR	Number of the error
T_MSG	Text of the message

Local Log

Messages are always logged to the application server's Event Log; however this is not necessarily true for the database as noted by the optional parameter passed to LogMessage, lLoggingOptions. An administrator with the appropriate access rights can connect to the MTS application server remotely and view its Event Log. Only one MTS package contains the Event Log Component, so that errors will all be written to the same application server Event Log.

Events logged via Visual Basic always have "VBRuntime" as the source. The Computer field is automatically populated with the name of the computer that is logging the event (i.e., the MTS application server) rather than the computer that
5   generated the event (typically a client computer).

The same event details that are written to the database are formatted into a readable string and written to the log. The text "The VB Application identified by ... Logged:" is automatically added by VB; the text that follows contains the
10   details of the message.

DATA ACCESS

15   All but a few exceptional cases use the "ExecuteQuery" API. This API covers singular database operations in which there exists a single input and a single output. Essentially should only exclude certain batch type operations.

The Data Access Framework serves the purposes of performance, consistency,
20   and maintainability.

Performance

The "ExecuteQuery" method incorporates usage patterns for using ADO in an efficient manner. Examples of these patterns include utilization of disconnected
25   recordsets, and explicitly declaring optional parameters which result in the best performance.

Consistency

This method provides a common interface for development of data access. Given
30   a simple and stable data access interface, best practices can be developed and disseminated.

Maintainability

Since the method is located in a single location, it is very modularized and can be maintained with little impact to its callers.

5

Application servers often use the ActiveX Data Objects (ADO) data access interface. This allows for a simplified programming model as well as enabling the embodiments to utilize a variety of data sources.

10

The "ExecuteQuery" Method

Overview

The "ExecuteQuery" method should be used for most application SQL calls.

15   This method encapsulates functionality for using ADO in a effective and efficient manner. This API applies to situations in which a single operation needs to be executed which returns a single recordset object.

Syntax

20

```
Set obj =  ExecuteQuery(vMsg, nTranType, sSQL, [nMaxRows], [adoTransConn],
[args])
```

Parameters

25

vMsg

This parameter is the TechArch struct. This is used as a token for information capture such as performance metrics, error information, and security.

30

nTranType

An application defined constant which indicates which type of operation is being performed. Values for this parameter can be one of the following constants:

35   cmSelect
          cmSelectLocal

cmUpdate
cmInsert
cmDelete

5   sSQL
String containing the SQL code to be performed against the DBMS.

nMaxRows (Optional)
Integer value which represent the maximum number of records that the

10   recordset of the current query will return.

adoTransConn (Optional)

An ADO Connection object. This is created and passed into execute

query for operations which require ADO transactional control (see "Using

15   Transactions" section)

args (Optional)
A list of parameters to be respectfully inserted into the SQL
statement.

20

Implementation

In one embodiment of the present invention the "ExecuteQuery" method resides

within the MservArch.bas file. This file should be incorporated into all

25   ServerComponent type projects. This will allow each server component access to

this method.

Note: Since this method is a public method in a "bas" module, it is globally

available from anywhere in the project.

30

```
        Public Function ExecuteQuery(vMsg As Variant, _
                                     nTranType As TranTypes, _
                                     sSQL As String, _
                                     Optional nMaxRows As Integer = 0, _
35                                   Optional adoTransConn As ADODB.Connection, _
                                     Optional colArguments As CCollection) As Variant
            On Error GoTo ErrorHandler
            Const cmMethodName As String = "ExecuteQuery"

40          StartTimeLogger vMsg, cmTimerIdDBTotal, cmClassName, cmMethodName
```

```
                        'find out if this call is an isolate operation or
                        'part of an ADO (not MTS) transaction
                        Dim isAtomicTrans As Boolean
        5               isAtomicTrans = adoTransConn Is Nothing

                        Dim nRecordsAffected As Integer
                        Dim adoRS As New ADODB.Recordset
                        Dim adoConn As ADODB.Connection
       10               Dim lAuxErrNumber As Long

                        'open a new connection or keep using the passed in connection
                        Set adoConn = IIf(isAtomicTrans, New ADODB.Connection, adoTransConn)

       15               If isAtomicTrans Then
                            adoConn.Open cmODBC_Connect

                            'ADO will wait indefinitely until the execution is complete during
                    performance
       20                   testing
                            #If IsPerfTest Then
                                adoConn.CommandTimeout = 0
                            #End If
                        End If
       25               'Make sure date args are formatted for DB2 if appropriate
                        If Not colArguments Is Nothing Then _
                            Set colArguments = FormatArgsForDB2(colArguments)

                        'merge the passed in arguments with the SQL string
       30               sSQL = MergeSQL(sSQL, colArguments)

                        Debug.Print Time & ": " & sSQL

                        'execute the SQL statement depending on the transaction type
       35               Select Case CStr(nTranType)
                            Case cmSelect
                                adoRS.MaxRecords = nMaxRows
                                adoRS.CursorLocation = adUseClient
                                adoRS.Open sSQL, adoConn, adOpenForwardOnly, adLockReadOnly,
       40           adCmdText
                                Set adoRS.ActiveConnection = Nothing
                                Set ExecuteQuery = adoRS
                            Case cmSelectLocal
                                adoRS.MaxRecords = nMaxRows
       45                       adoRS.CursorLocation = adUseClient
                                adoRS.Open sSQL, adoConn, adOpenStatic, adLockBatchOptimistic,
                   adCmdText
                                Set adoRS.ActiveConnection = Nothing
                                Set ExecuteQuery = adoRS
       50               Case cmInsert
                                Set adoRS = adoConn.Execute(sSQL, nRecordsAffected, adCmdText)
                                If nRecordsAffected <= 0 Then Err.Raise cmErrQueryInsert
                                Set adoRS = Nothing
```

```
                    ExecuteQuery = nRecordsAffected
                Case cmUpdate, cmDelete
                    Set adoRS = adoConn.Execute(sSQL, nRecordsAffected, adCmdText)
                    If nRecordsAffected <= 0 Then Err.Raise cmErrOptimisticLock
5                   Set adoRS = Nothing
                    ExecuteQuery = nRecordsAffected
                Case cmSpFileNote
                    Set adoRS = adoConn.Execute(sSQL, nRecordsAffected, adCmdText)
                    Set adoRS = Nothing
10              Case Else
                    Err.Raise cmErrInvalidParameters
            End Select

            StopTimeLogger vMsg, cmTimerIdDBTotal, cmClassName, cmMethodName
15          Exit Function

    ErrorHandler:
            Dim objArch As Object
            Set objArch = CreateObject("cmArch.CTechArch")
20
            Select Case CStr(Err)
                Case cmErrQueryInsert, cmErrOptimisticLock, cmErrInvalidParameters
                    'Raise error
                    Err.Raise Err
25
                Case cmErrDSNNotFound
                    Dim sMsgText As String
                    sMsgText = "Data Source Name not found." & vbCrLf & "( " & _
                                CStr(objArch.RegMan.GetServerDSN) & " )"
30
                    ' Create a new message log and log the message
                    objArch.LogMan.LogMessage vMsg, cmSeverityFatal, cmClassName,
    cmMethodName,
                                GetVersion(), cmErrDSNNotFound, sMsgText,
35  cmLogToEventViewerOnly


                    lAuxErrNumber = adoConn.Errors(0).NativeError 'The error code is
    stored since
40                                                                  when closing the
    conection it will
                                                                    be lost
                    If adoConn.State <> adStateClosed Then adoConn.Close

45                  Err.Raise cmErrDSNNotFound, , sMsgText
                Case Else
                    ' Create a new message log and log the message
                    objArch.LogMan.LogMessage vMsg, cmSeverityFatal, cmClassName,
    cmMethodName,
50                              GetVersion(), Err.Number, Err.Description,
    cmLogToEventViewerOnly
```

```
            lAuxErrNumber = adoConn.Errors(0).NativeError 'The error code is
stored since
                                                      when closing the
conection it will
                                                      be lost
            If adoConn.State <> adStateClosed Then adoConn.Close
            Err.Raise lAuxErrNumber
      End Select
End Function
```

Selecting Records

ExecuteQuery utilizes disconnected recordsets for "Select" type statements. This requires that the clients, particularly the CCA's contain a reference to ADOR, ActiveX Data Object Recordset. This DLL is a subset of the ADODB DLL. ADOR contains only the recordset object.

Using disconnected recordsets allows marshalling of recordset objects from sever to client. This performs much more efficiently than the variant array which is associated with using the "GetRows" API on the server. This performance gain is especially apparent when the application server is under load of a large number of concurrent users.

Sample from Client Component Adapter (CCA)

```
Dim vAns as Variant
Dim adoRS As ADOR.Recordset
Set adoRS = objServer.PerformSelect(vMsg, nId)
If objRS.EOF Then
    Set objRS = Nothing
    Exit Function
End If

vAns = adoRS.GetRows
Set adoRS = Nothing

'Marshall vAns into objects
...
```

Sample from Server Component

```
Private Const cmCustSQL = "Select * from Customer where id = ?"
```

```
Public Function PerformSelect(vMsg, nId) as Variant
    Dim colArgs as CCollection
    Set colArgs = New Ccollection
    colArgs.Add nId
    Set PerformSelect = ExecuteQuery(vMsg, cmSelect, sCustSQL, , ,colArgs)
End Function
```

Code Clip from ExecuteQuery (Select Section)

```
Case cmSelect
    adoRS.MaxRecords = nMaxRows
    adoRS.CursorLocation = adUseClient
    adoRS.Open sSQL, adoConn, adOpenForwardOnly, adLockReadOnly, adCmdText
    Set ExecuteQuery = adoRS
    . . .
```

Inserting Records

Inserting records requires certain information pertaining to optimistic locking. On the server a unique value is requested to indicate the last time modified. This unique value is returned back to the requestor such that it can be used to later database operations.

Sample from Client Component Adapter (CCA)

```
Dim vNewTS as Variant
vNewTS = objServer.PerformInsert(vMsg, nId, sName)
'Set object's TimeStamp to vNewTS
```

Sample from Server Component

```
Private Const cmCustInsertSQL = "Insert Customer (nId, Name, LastUpdated)
Values(?, '?', ?)"

Public Function PerformInsert(vMsg, nId, sName) As Variant
Dim lCurrTS as Long
lCurrTS = GetTimeStamp
Dim colArgs as CCollection
Set colArgs = New Ccollection
colArgs.Add nId
colArgs.Add sName
colArgs.Add lCurrTS
ExecuteQuery(vMsg, cmInsert, sCustInsertSQL, , , colArgs)
PerformInsert = lCurrTS
```

Code Clip from ExecuteQuery (Insert Section)

```
Case cmInsert
    Set adoRS = adoConn.Execute(sSQL, nRecordsAffected, adCmdText)
5   If nRecordsAffected <= 0 Then Err.Raise cmErrQueryInsert
    Set adoRS = Nothing
    ExecuteQuery = nRecordsAffected
```

10

Updating Records

Updating records requires certain information pertaining to optimistic locking.
On the server a unique value is requested to indicate the last time modified. Also
15 the last read timestamp is used to validate, during the update, that the record has
not been modified since last time read.

Sample from Client Component Adapter (CCA)

```
Dim vNewTS as Variant
20  vNewTS = objServer.PerformUpdate(vMsg, 1, 'Rick', 8907654)
    'Set object's TimeStamp to vNewTS
```

Sample Code Clip from Server Component

```
Private Const cmCustUpdateSQL = _
25  "Update Customer Set Name = '?', LastUpdated = ? " & _
    "Where Id = ? " & _
    "And LastUpdated = ? "

Public Function PerformUpdate(vMsg, nId, sName, lLastTS) As Variant
30      Dim lCurrTS as Long
        lCurrTS = GetTimeStamp
        Dim colArgs as CCollection
        Set colArgs = New Ccollection
        colArgs.Add sName
35      colArgs.Add lCurrTS
        colArgs.Add nId
        colArgs.Add lLastTS
        PerformUpdate = ExecuteQuery(vMsg, cmUpdate, sCustUpdateSQL, , , colArgs)
        PerformUpdate = lCurrTS
40  End Function
```

Code Clip from ExecuteQuery (Update Section)

```
Case cmUpdate
    Set adoRS = adoConn.Execute(sSQL, nRecordsAffected, adCmdText)
45  If nRecordsAffected < 0 Then Err.Raise cmErrOptimisticLock
    ExecuteQuery = nRecordsAffected
```

Deleting Records

In deleting records the last read timestamp is used to validate, during the delete, that the record has not been modified since last time read.

Sample from Client Component Adapter (CCA)

```
Dim vAns as Variant
vAns = objServer.PerformDelete(vMsg, nId ,lLastTS)
```

Sample from Server Component

```
Private Const cmCustDeleteSQL = _
"Delete From Customer " & _
"Where Id = ? " & _
"And LastUpdated = ? "

Public Function PerformDelete(vMsg, nId lLastTS) As Variant
    Dim colArgs as CCollection
    Set colArgs = New Ccollection
    colArgs.Add nId
    colArgs.Add lLastTS
    PerformDelete = ExecuteQuery(vMsg, cmDelete, cmCustDeleteSQL)
Exit Function
```

Code Clip from ExecuteQuery (Delete Section)

```
Case cmDelete
    Set adoRS = adoConn.Execute(sSQL, nRecordsAffected, adCmdText)
    If nRecordsAffected < 0 Then Err.Raise cmErrOptimisticLock
    ExecuteQuery = nRecordsAffected
```

DATABASE LOCKING FRAMEWORK

Database Locking ensures the integrity of the database in a multi-user environment. Locking prevents the common problem of lost updates from multiple users updating the same record.

Solution Options

Pessimistic Locking

This policy of locking allows the first user to have full access to the record while

5   following users are denied access or have read only access until the record is
unlocked. There are drawbacks to this method of locking. It is a method that is
prone to deadlocks on the database as well poor performance when conflicts are
encountered.

Optimistic Locking

10   The optimistic approach to record locking is based on the assumption that it is not
normal processing for multiple users to both read and update records
concurrently. This situation is treated as exceptional processing rather than
normal processing. Locks are not actually placed on the database at read time. A
timestamp mechanism is used at time of update or delete to ensure that another

15   user has not modified or deleted the record since you last read the record.

A preferred embodiment of the present invention uses an optimistic locking
approach to concurrency control. This ensures database integrity as well as the
low overhead associated with this form of locking. Other benefits to this method

20   are increased availability of records to multiple users, and a minimization of
database deadlocks.

Table candidates for concurrency control are identified during the "Data
Modeling Exercise". The only table which is updated concurrently is the

25   Optimistic Locking mechanism. Once these are identified, the following is added
to the application.

Add "N_Last_Updt" field to table in database;

Error Handling routines on those operations which modify or delete from this table; and

Display/Notification to user that the error has occurred.

Usage

The chart below describes the roles of the two basic types of components to enable optimistic locking.

5   Assumption: The optimistic locking field is of type Date and is named "N_Last_Updt"

	Client Components	Server Components
Read Access	Store N_Last_Updt value in the business object for use in possible updates or deletes.	Retrieve data (Always including N_Last_Updt field). SELECT Id, FirstName, N_Last_Updt FROM Customer WHERE id = 10;
Inserts	Normal	Dim lCurrTS As Double lCurrTS = GetTimeStamp INSERT INTO Customer (Id, FirstName, N_Last_Updt) VALUES (1, "Rick", lCurrTS); Return new timestamp (lCurrTS) as well as new Id

Updates	Pass previously read timestamp to identify whether row was modified. This is in addition to a unique identifier and whatever data needs to be updated.
	
	Handle exception if record has been previously modified.
	      Notify user of conflict.
	      Rollback any changes.
		Dim lCurrTS As Double
		lCurrTS = GetTimeStamp
		
		UPDATE Customer
		SET firstName = "Richard",
		            N_Last_Updt = lCurrTS
		WHERE id = 1
		AND LastUpdate = lastReadTimestamp;
		
		If no rows are affected, handle and propagate error back out to the client.
		
		Return new timestamp (lCurrTS)

Deletes	Pass previously read timestamp to identify whether row was modified. This is in addition to a unique identifier
	
	Handle exception if record has been previously modified.
	      Notify user of conflict.
	      Rollback any changes.
		DELETE Customer
		WHERE id = 1
		AND N_Last_Updt = lastReadTimestamp;
		
		If no rows are affected, handle and propagate error back out to the client.

10

LARGE RESULT SET

When retrieving records from a database, if the search criteria is too broad, the amount of data required to be retrieved from the database and passed across the network will affect user perceived performance. Windows requesting such data will be slow to paint and searches will be slow. The formation of the database queries is made such that a workable amount of data is retrieved. There are a few options for addressing the problems that occur from large result sets. The options are given below in order of preference.

Redesign the interface/ controller to return smaller result sets. By designing the controllers that present the database queries intelligently, the queries that are presented to the database server do not return a result set that is large enough to affect user perceived performance. In essence, the potential to retrieve too many records indicates that the UIs and the controllers have been designed differently. An example of a well designed Search UI is one where the user is required to enter in a minimum search criteria to prevent an excessively large result set.

Have Scrollable Result Sets. The scrolling retrieval of a large result set is the incremental retrieval of a result subset repeated as many times as the user requests or until the entire result set is obtained. Results are retrieved by the Bounded Query Approach where the first record is determined by a where clause with calculated values.

Scrollable Result Set Client requirements

Preferred UI
The preferred displays are as follows:

Returned results are displayed in a GreenTree List Box;

An action button with the label **More...** is provided for the user to obtain the remaining results;

5   The More button is enabled when the user has performed an initial search and there are still results to be retrieved;

The More button is disabled when there are no more results to retrieve;

10   The List Box and the Action button is contained within a group box to provide a visual association between the button and the List Box.

Bounded Query

15   Queries that are implemented with the limited result sets are sent to the server. The server implements the executeQuery method to retrieve the recordset as usual. Limited result queries have an order by clause that includes the business required sort order along with a sufficient number of columns to ensure that all rows can be uniquely identified. The recordset is limited by the nMaxRows

20   variable passed from the client incremented to obtain the first row of the next result set. The return from the component is a recordset just the same as with a query that is not limited. The CCA 208 creates the objects and passes these back to the controller 206 . The Controller 206 adds this returned collection of object to its collection of objects (an accumulation of previous results) and while doing

25   so will performs the comparison of the last object to the first object of the next row. The values necessary to discriminate the two rows are added to the variant array that is necessary to pass to the component for the subsequent query.

The Controller 206 on the client retains the values for nMaxRows, the initial SQL

30   statement, and array of values to discern between the last row of the previous query and the first row of the next query. The mechanism by which the controller

206 is aware that there are more records to retrieve is by checking the number of results is one greater than the max number of rows. To prevent the retrieval of records past the end of file, the controller 206 disables these functions on the UI. For example, a command button **More** on the UI, used to requested the data, is

5   disabled when the number of objects returned is less than nMaxRows + 1 .

Application responsibility

Server

10   The Server component is responsible for creating a collection of arguments and appending the SQL statement to add a where clause that will be able to discriminate between the last row of the previous query and the first row of the next.

CCA

15   The CCA 208 processes the recordset into objects as in non limited queries. The CCA 208 forwards the variant array passed from the Controller 206 to identify the limited results.

Controller

20   The controller 206 has the responsibility of disabling the **More** control when the end of file has been reached. The controller 206 populates the variant array (vKeys) with the values necessary to determine start of next query.

Example

25   A CCA 208 is coded for a user defined search which has the potential to return a sizable result set. The code example below implements the Bounded Query approach.

30

On the Server the developer codes the query as follows:

```
     Public Function RetrieveBusinessObjects(vMsg As Variant, ByVal sSql As String,
     ByVal nMaxRows As Integer, Optional ByVal vKeys As Variant) As Recordset
         On Error GoTo ErrorHandler

5        'Declare local constants
         Const cmMethodName As String = "RetrieveBusinessObjects"


         'Declare local variables
10       Dim cmClassName As String
         Dim colArgs As New CCollection

         'initialize instance variables
         cmClassName = "CSRSTestComp"
15

         'fill argument collection
         Set colArgs = ArgumentsForBusinessObject(vKeys , sSQL)

20       'increment nMaxRows to obtain row for comparison

         nMaxRows = nMaxRows + 1


         'ExecuteQuery
25           Set RetrieveBusinessObjects = ExecuteQuery(vMsg, cmSelectLocal, sQuery,
     nMaxRows, , colArgs)


         'Tell MTS we're done
30       GetObjectContext.SetComplete
         Exit Function

     ErrorHandler:
         Select Case Err.Number
35
             Case Else
                 Dim iResumeCode As Integer
                 iResumeCode = GeneralErrorHandler(vMsg, cmServer, cmClassName,
     cmMethodName)
40               Select Case iResumeCode
                     Case cmErrorResume
                         Resume
                     Case cmErrorResumeNext
                         Resume Next
45                   Case cmErrorExit
                         Exit Function
                     Case Else
                         GetObjectContext.SetAbort
                         Err.Raise Err.Number
50               End Select
         End Select
     End Function
```

To determine the additional where clause necessary to determine the starting

55 point of the query, the following method is added:


```
     Private Function ArgumentsForBusinessObject(vKeys As Variant, sSql As string )
     As CCollection
         Dim colArgs As Ccollection
60       Const cmGreaterThanWhereString As String = " ? > ? "
         Const cmGreaterThanOrEqualWhereString As String = " ? >= ? AND "

         ' initialize local variables
         Set colArgs = New Ccollection
```

```
        sSql = sSql + "WHERE"


        With colArgs
5           If vKeys(0) <> Empty Then
                .Add ("N_TASK_TEMPL_ID")
                .Add (vKeys(0))

            End If
10          'If vKeys(1) <> Nothing Then
                '.Add value2 fieldName
                '.add vKeys(1)
                sSql = sSql + cmGreaterThanOrEqualWhereString
            'End If
15          'If vKeys(2) <> Nothing Then
                '.Add value3 fieldName
                '.add vKeys(2)
                sSql = sSql + cmGreaterThanOrEqualWhereString
            'End If
20      End With


        'finalize SQL statement
        sSql = sSql + cmGreaterThanWhereString

25      Set ArgumentsForBusinessObject = colArgs
    End Function
```

On the CCA 208, allowance must be made for the passing of the vKeys

Public Function RetrieveBusinessObjects(vMsg As Variant, sSql As String, nMaxRows As Integer, Optional ByVal vKeys As Variant) As CCollection

```
                .
        Set percmpComponent = New CSRSTestComp
35      Dim i As Integer

        Set adoRS = percmpComponent.RetrieveBusinessObjects(vMsg, sSql, nMaxRows,
    vKeys)

40      'convert recordset to business objects
        adoRS.MoveFirst
        Do Until adoRS.EOF
            Call ConvertToBusinessObject
            adoRS.MoveNext
45      Loop

        'return the collection of business objects
        Set RetrieveBusinessObjects = dictBusinessObject
        Set dictBusinessObject = New CCollection
50
```

```
End Function
```

5   The controller initiates the query and updates the variant array of keys and form

204 properties based on the return. In addition to the code shown for the example

below, the More Control is enabled if the search is cleared.

```
      'declare instance variables
10    Private nMaxRows As Integer
      Dim interimResults As CCollection
      Dim vResults As CCollection
      Dim vKeys(3) As Variant

15    'declare Constants
      Private Const nDefaultAmount As Long = 50
      Private Const cmRetrieveBusinessObjectSQL = "SELECT * FROM NODE_RULE ORDER BY
      _ N_TASK_TEMPL_ID"

20
```

During class initialization perform the following:

```
      Public Sub Class_init()

25        'obtain settings from registry
          nMaxRows = CInt(GetSetting(cmRegApp, cmRegArchSection,
      cmLimitedResultAmountKey, lDefaultAmount))

          Call resetSearch
30        Set objCCA = New {CCA class name}

      End Sub
```

Search reset functionality is kept outside of initialization so this may be called

35   from other parts of the application.

```
      Public Sub resetSearch()
          Dim I as Integer
              Set vResults = New Ccollection
              For I = 0 To 3
40                Set vKeys(I)  = Empty
              Next
      Set vKeys(0)  = Empty
              frmCurrentForm.cmdMore.Enabled = True

45    End Sub

      Public Sub RetrieveBusinessObjects()
              Const cmMethodName As String = "retrieveBusinessObjects"
```

```
                    Call RetainMouse

                    ' get arch message
                    Dim vMsg As Variant
 5                  vMsg = objApp.objArch.AsMsgStruct()

                    ' call the component
                    Dim pair As CArchPair

10                  'Declare local variables
                    Dim sSql As String
                    Dim colArgs As CCollection
                    Dim cmClassName As String

15              Set interimResults = objCCA.RetrieveBusinessObjects(vMsg,
            cmRetrieveBusinessObjectSQL, nMaxRows, vKeys)

                ctr = ProcessObjectCollection

20              'stop if size of return is less than the maximum
                    If ctr < nMaxRows + 1  Then frmCurrentForm.cmdMore.Enabled = False
                    ' restore pointer
                    Screen.MousePointer = lPrevPtr
                End Sub
25
```

In order to retain the values to discriminate between the last row of the result set
and the first row of the next the following method on the controller is used:

```
            Private Function ProcessObjectCollection() As Integer
30
            ' merge results with the instance variable for the collection
                    Dim ctr As Integer
                        ctr = 0
                    For Each element In interimResults
35                      ctr = ctr + 1
                        'retain Keys for subsequent Queries
                        With element
                        Select Case ctr
                            Case nMaxRows
40                              'store all values that may be used for row comparison
                                vKeys(0) = .NodeId

                                'add last object to collection
                                vResults.Add element
45
                            Case nMaxRows + 1
                                'last object only used for comparison
                                'If the proceeding value can be used to uniquely
                                'identify row then delete value from array
50                              ' THERE SHOULD BE N - 1 nested If statements where N =
            size of vKeys
```

```
'If .value2 <> vKeys(1) Then
    'vKeys(2) = Empty
    If .NodeId <> vKeys(0) Then vKeys(1) = Empty
'End If
            Case Else
                vResults.Add element

            End Select
            End With
        Next
    ProcessObjectCollection = ctr
End Function
```

Operation of example with data

Person

First Name	Last Name	Status	Unique ID
Joy	Andersen	Closed	22
Jay	Anderson	Open	12
John	Barleycorn	Closed	512
John	Barleycorn	Open	32
Esther	Davidson	Open	88
David	Dyson	Closed	98
Bobby	Halford	Open	234
Steven	Jackowski	Closed	4
Kyle	Johnsen	Open	65
Jeff	Johansen	Open	13
Mary	Johnson	Closed	24
Larry	Olsen	Open	21
William	O'Neil	Closed	29
Jane	Pick	Open	3285

For this example let nMaxRows = 3. The business case calls for the result set to be ordered by the last name, and developer knows that any row can be uniquely identified by the FirstName, LastName, and Unique ID fields so the initial SQL added as a constant in the controller should be;

```
SELECT * FROM Person ORDER BY  LastName, FirstName,
Unique_ID
```

Initial Query

The first query is sent with an empty vKeys Array. When the server receives this query, the method **ArgumentsForBusinessObject** identifies the elements as

being empty and does not populate the colArgs. The query is executed with the intial SQL unchanged. The recordset of size nMaxRows + 1 is returned to the CCA 208 and processed the same as non-limited results. The CCA 208 returns the collection of objects to the controller 206. The controller 206 proceeds to

5   populate the vResults collection with the returned objects. vResults is the comprehensive collection of objects returned. When the last object of the first request is reached (at nMaxRows), the values are stored in vKeys as such;

vKeys(0) = LastName (Barleycorn)

vKeys(1) = FirstName (John)

10 vKeys(2) = Unique_ID (512)

When the First Object of the next request is reached (at nMaxRows +1), comparison of the object variables against the vKeys values is performed. Because the last names match, vKeys(2) will not be deleted and no further checks are performed.

15 *Subsequent Query*

The subsequent query will pass vKeys along with it. The server creates the collection of arguments from vKeys and append the sSql string in accordance. The sSql statement that is passed to execute query is

```
SELECT * FROM Person ORDER BY  LastName, FirstName,
```

20
```
Unique_ID WHERE ? >= ? AND ? >= ? AND ? > ?
```

This sSql and collection is included in the call to **ExecuteQuery** which merges the arguments with the string relying on the architecture method MergeSQL to complete the SQL statement.

The starting point of the recordset is defined by the WHERE clause and the limit

25 is set by the nMaxRows value.

Query less restrictive WHERE criteria

After the second query the last row of the query is David Dyson and the next is Bobby Halford. Because the last name is different, vKeys will be empty except

30 for vKeys(0) = Dyson.

The **ProcessObjectCollection** will populate vKeys as follows when processing nMaxRows object:

vKeys(0) = LastName (Dyson)

vKeys(1) = FirstName (David)

5   vKeys(2) = Unique_ID (98)

After identifying the differences between vKeys values and the nMaxRows + 1 object the vKeys array is updated as follows:

vKeys(0) = LastName (Dyson)

10   vKeys(1) = Empty

vKeys(2) = Empty

The query that is returned from **ArgumentsForBusinessObject** is

```
SELECT * FROM Person ORDER BY  LastName, FirstName,
```
15   `Unique_ID WHERE ? > ?`

and the colArgs possessing the fieldname FirstName and the value ("David").

ExecuteQuery merges the arguments with the sql statement as before and returns the value.

20   *Ending*

          After the fifth iteration the result set will only possess 2 records. When the controller 206 processes the returned collection the counter returned from **ProcessObjectCollection** is less than nMaxRows + 1 which indicates that all records have been retrieved.

25

30

SECURITY FRAMEWORK

Implementation

Figure 8 shows a representation of the Security Framework 800 and its main components.

It can be seen from Figure 8 that the Security object 802 is present at the Client and a Security API is provided at the server. The Security object 802 provides one method responsible for authorizing any operation, being given the vMsg structure, an operation ID and an optional parameter describing the operation's context.

Client

User Authentication:

User authentication is handled via a method located in the Security object 802 called *IsOperAuthorized*. As the Application object loads, it calls the *IsOperAuthorized* method, with the operation being "Login", before executing further processing. This method subsequently calls a authentication DLL, which is responsible for identifying the user as an authorized user within the Corporate Security.

UI Controllers:

The UI Controllers limit access to their functions by restricting access to specific widgets through enabling and disabling them. The logic for the enabling and disabling of widgets remains on the UI Controller 206, but the logic to determine whether a user has access to a specific functionality is located in the Security object 802 in the form of business rules. The UI Controller 206 calls the *IsOperAuthorized* method in order to set the state of its widgets.

Server

Server security is implemented by restricting access to the data in three different

5   ways:

Server Security Method

Server Components 222 call the *IsOperAuthorized* API in the Architecture before

executing every operation. In all cases the Security object 802 returns a boolean,

10   according to the user's access rights and the business rules

SQL Filtering

Includes security attributes, like claim sensitiveness or public/private file note,

into the SQL statements when selecting or updating rows. This efficiently

15   restricts the resulting data set, and avoids the return of restricted data to the client.

Description

20   Any GUI related security is implemented at the Client using the Security object

802. The information is available both at the Client Profile and Business Objects

207 which enables the security rules to be properly evaluated.

IsOperAuthorized is called to set widgets upon the loading of a UI or if there is a

25   change of state within the UI.

User authentication always is used by the Application Objects 202 in order to

validate user privilege to launch the application.

30   SQL Filtering is used in the cases where sensitive data must not even be available

at the Client, or where there is a great advantage on reducing the size of the data

set returned to the Client.

SQL Filtering is only used in very rare cases where performance is a serious concern. It is used carefully in order to avoid increased complexity and performance impacts because some queries can be cumbersome and embedding security on them could increase complexity even more.

5

Security Framework

10 *Overview*

The Security object 802 serves the purpose of holding hard coded business rules to grant or deny user access for various application functions. This information is returned to the UI controllers 206 which make the necessary modifications on the UI state. The ClientProfile object serves the purpose of caching user specific

15 (and static) security information directly on the client. This information is necessary to evaluate the business rules at the Security object 802.

<u>Relationships</u>

20

Figure 9 shows the relationships between the security element and other elements.

Architecture Object

25 The TechArch object is responsible for providing access and maintaining the state of the ClientProfile 902 and Security objects 802. The ClientProfile object 902 is instantiated and destroyed in the TechArch's initialization and terminate methods, respectively. This object is maintained through an instance variable on the TechArch object.

30

CInitCompCCA

The CInitCompCCA object 904 provides two services to the architecture object 200, it serves as an access point to the CInitComp Server 906, and it Marshalls the query result set into a ClientProfile object 902.

5

CInitComp

The CInitComp server object 906 provides data access to the data that resides in the organization tables 908. This data is useful on the client to determine level of access to data based on hard coded business rules.

10

Organization Tables

The Organization tables 908 contain user, employee and unit information necessary to build the hierarchy of information necessary to determine level of access to sensitive information.

15

Client Profile

The ClientProfile object 902 serves the purpose of caching static, user specific security information directly on the client. This information is necessary to determine data access level of information to the user, which is accomplished by
20 passing the necessary values to the Security object 802.

Security Object

The Security Object 802 contains business rules used to determine a user's access privileges in relation to specific functions. The object accepts certain parameters
25 passed in by the various UI Controllers 206 and passes them to through the business rule logic which, in turn, interrogates the Client Profile object 902 for specific user information.

Client Profile

Attributes

5   The following are internal attributes for the Client Profile object 902. These
attributes are not exposed to the application and should only be used by the
Security object 802:

- sProfile:

10

This attribute is passed by the legacy application at start-up and contains the
user's TSIds, External Indicator, Count of Group Elements and Group
Elements. It is marshalled into these attributes by request of the application
objects.

15

- colSpecialUsers:

This attribute caches information from a table containing special users which
do not fit into one of the described roles, such as Organization Librarian.

20   (e.g., Vice President or CEO of the corporation.)

- sTSId:

This is the current users' TSId, and it corresponds to his/her Windows NT Id.

25   It is used to get information about the current logged on user from the
Organizational Tables 908.

30

- sEmployeeId:

  This corresponds to the user's employee Id, as stored in the Organizational
  tables 908. It is used against the passed in employee Id, in order to check
  5   relationship between performers and the current user.

- sEmployeeName, sEmployeeFirst, sEmployeeMI and sEmployeeLast:

  All these attributes correspond to the current user's name.

  10

- dictClientPrivileges:

  This attribute contains a collection of identifiers that indicate what
  role/authority an individual plays/possesses. This value is used to identify the
  15   static role of the logged in user.

  These values are used for security business logic which grants or denies
  access based on whether the user is internal or external, or whether the user is
  in a given administrative role. Existing values are the following:
  20
  - SC - Indicates sensitive Claim authority
  - CC - Indicates Change Claim status authority
  - MT - Indicates maintain F&C Templates authority
  - MO - Indicates maintain Organization authority
  - MR - Indicates maintain Roles authority

  25

  The following are the proposed additions:
  - TA - Indicates authority to execute Task Assistant
  - FN - Indicates authority to execute FileNotes
  - CH - Indicates authority to execute Claim History
  30   - TL - Indicates authority to maintain Task Templates

- dictProxyList:

5 This attribute contains an employees' reporting hierarchy. It is used to determine whether the current user/employee has permission to perform some action based on his/her relationship to other users/employees within their hierarchy. A business example of this is the case of a supervisor, who has rights to view information that his/her subordinates have access to. The

10 relationship API's make use of dictProxyList to determine if the user assigned to the information is super or subordinate of the current user.

- boolInternal:

15 This attribute indicates whether the logged in user is external or internal. It is also marshalled from the sProfile attribute, passed in by the legacy application.

Public Methods

20 The following are the APIs exposed by the Client Profile object. These APIs are used for security checking by the Security object and should not be used by the developers in any portion of the application.

- GetAuthorizedEmployees As Collection

25

This function returns a collection of employee Ids from the employees supervised by the current user.

- IsSuperOf(sUserId) As Boolean

30

This API returns true if the logged in user is a super of the passed in user Id. It looks up the sUserId value inside the dictProxyList attribute.

- IsRelativeOf(sUserId) As Boolean

  This API returns true if the passed in user Id corresponds to either the logged in user or someone from the dictProxyList.

- IsInternal As Boolean

  This API is used to grant or restrict the user to information based on whether the data is private to the organization whether the user is internal or external.

- IsInRole(sRole) As Boolean

  This API looks up the appropriate sRole value contained within the dictClientRoles attribute to determine whether the current user is authorized to perform that role.

The following accessors are used to get data from the Client Profile's object:

- UserId: returns sTSId .
- EmployeeId: return sEmployeeId
- EmployeeName: returns sEmployeeName
- EmployeeFirstName: returns sEmployeeFirst
- EmployeeLastName: returns sEmployeeLast
- EmployeeMiddleInitial: returns sEmployeeMI
- ExpandTree: returns boolExpandTreePreference
- TemplatePathPreference: returns sTemplatePathPreference

Security Object

Public Methods

The following API is exposed by the Security Object and is used by the application for security checking:

5

- IsOperAuthorized(vMsg As Variant, nOperations As cmOperations, vContext As Variant) as Boolean

This API will return true or false depending on what is returned from the business rule functions to determine user access levels. This API is called on two situations:

1. When setting the initial state before loading the form. If a security requirement exists, IsOperAuthorized is called for the appropriate operation.
2. After any relevant change on the UI state. For example, when a sensitive claim is highlighted on the Task Assistant window. A relevant change is one which brings the need for a security check.

20 The valid values for the enumeration and the correspondent context data are:

- cmMaintainFormsCorr (none)
- cmRunEventProcessor (none)
- cmWorkOnSensitiveClaim (a Claim object)
- cmMaintainPersonalProfile (none)
- cmMaintainWorkplan (none)
- cmDeleteFileNote (a File Note object)
- cmMaintainTaskLIbrary (none)
- cmMaintainOrg (none)

Server Security APIs

5 • *IsSVCOperAuthorized(vMsg As Variant, sOperations As String , vContext As Variant) as Boolean* –

   This API is called by every method on the server that persists data or can potentially access sensitive data (reactive approach).

10

   • *IsOperAuthorized(vMsg As Variant, nOperations As cmOperations, vContext As Variant) as Boolean*

   This API is available for those cases where a proactive security check is
15 needed on the server.

Implementation Examples

20
   The following examples show some ways to implement the options described above:

   ### *Client*

25
   • <u>Business Logic</u>

   IsOperAuthorized

30 Let's consider the case of the Task Assistant window, where the user should not be allowed to view any information on a sensitive claim if he/she is not

the claim performer or the performer's supervisor. The following code would be at the Controller:

```
Private Sub TaskTree_NodeChanged(....)
    myController.SetCurrentTask
    myController.SetState
End Sub

Private Sub SetState()
    Dim objSecurity as Object
    Dim vContext(1) as Object

    Set objSecurity = taaApp.taoArch.objSecurity

    vContext(0) = CurrentClaim
    vContext(1) = CurrentTask

    tlbEditIcon.Enabled =
                objSecurity.IsOperAuthorized(vMsg, cmWorkOnSensitiveClaim,
vContext)

End Sub
```

Let's consider the case of the Maintain Correspondence Search window where only a user who is a Forms and Correspondence Librarian should be allowed to delete a template. The following code would be at the Controller:

```
Private Sub SetWindowMode()
    Dim objSecurity as Object
    Set objSecurity = taaApp.taoArch.objSecurity

    . . .

    tlbEditIcon.Enabled = objSecurity.IsOperAuthorized(vMsg,
cmMaintainFormsCorr)          .

End Sub
```

Server

- SQL Filtering:

Let's consider the example of the Draft File Note window, where a user can only look at the draft file notes on which he/she is the author. At the controller, one would have:

```
5       Public Sub GetDraftFNotes()
            Dim objCP as Object
            Set objCP = taoArch.objClientProfile

            Dim fntCCA as Object
10          Set fntCCA = taaApp.taoArch.GetCCA(cmCCAFileNote)

            Call fntCCA.GetADraftFNote(vMsg, objCP.sOrgUserId, colFNotes)
        End Sub
```

15   And at the Component, the SQL statement would be:

```
        Select  nFNoteId,
            sFNoteAuthor,
            dFNoteFinal,
20          :
            :
            From  File Note
            Where sFileNoteSts = 'D'
            And   sFNoteAuthor = sAuthor
25
```

Task Engine Application

This application runs on the server as a background process or service with no
30   direct interaction with Client applications, so it doesn't need any GUI related security. Basically, its main actions are limited to the generation of new tasks in response to externally generated events or, more specifically, it:

- Reads static information from the Task Template tables;
35   - Reads events from the Event tables;
- Inserts tasks on the Task table.

In this sense, its security is totally dependent on external entities as described below:

40

- The Task Library application is the entrance point for any changes on the Task Template database tables. It will make use of the options described above in order to fulfill its security requirements.

5
- Events are generated from legacy applications, so the Task Engine relies completely on the security implemented for these applications in order to control the generation of events.

- Another level of security for event generation relies on the Database authorization and authentication functions. Only authorized components have access to the database tables (this is valid for all the other
10
applications as well).

CLAIM FOLDER

Definition

The Claim Folder manages claim information from first notice through closing
15
and archiving. It does this by providing a structured and easy to use interface that supports multiple business processes for handling claims. The information that it captures is fed to many other components that allow claims professionals to make use of enabling applications that reduce their workload. Because physical claim files are still required, the claim folder provides capabilities that support physical
20
file tracking. It works with the LEGACY system to support all the capabilities that exist within the current system.

The primary processes supported by the Claim Folder are:

25
- First Notice of Loss

   The Claim Folder is the primary entry point for new loss information. Claim files exist in the Claim Folder before they are "pushed" to the LEGACY system to perform financial processing.

- Claim Inquiry

Claim Folder supports internal and external inquires for claim information. The folder design allows quick access to various levels of information within the claim for many different reasons.

- Initiation of Claim Handling

5 The Claim Folder provides initial loss information to the claim professional so they may begin the process of making first contacts with appropriate participants in the claim. It allows them to view and enter data received through their initial contacts and investigation.

- Investigation and Evaluation

10 The Claim Folder provides access to detailed information needed for the investigation and evaluation process. It allows the claim handler to navigate between all the applications and information they need to support these processes.

- Identifying Claim Events

15 The Claim Folder identifies critical events that occur in the life of a claim, such as a change of status, which can trigger responses in other components to perform automated functions, like triggering tasks in the Task Assistant.

- Managing the Physical File

20 The Claim Folder supports better tracking capabilities for the physical files that go along with the electronic record of a claim.

Value

25 By capturing detailed information on claims, the Claim Folder tries to improve the efficiency of claim professionals in many ways. First, because the information is organized in a logical, easy to use format, there is less digging required to find basic information to support any number of inquiries. Second, the Claim Folder uses its information to support other applications like Forms

30 and Correspondence, so that claim information does not have to be reentered

every time it is needed. Third, it provides better ways to find physical files to reduce the time required finding and working with them. Beyond this, there are many other potential uses of claim folder information.

5   The Claim Folder also tries to overcome some of the current processing requirements that the LEGACY system imposes such as recording losses without claims, requiring policy numbers for claim set-up, requiring reserves for lines, and other restrictions. This will reduce some of the low-value added work required to feed the LEGACY system.

10

Finally, the Claim Folder organizes and coordinates information on participants and performers so that all people involved in a claim can be identified quickly and easily.

15   *Key Users*

Although claim professionals are the primary users of the Claim Folder, any claims professional can utilize the Claim Folder to learn about a claim or answer an inquiry about a claim.

20   *Component Functionality*

Because the Claim Folder is the primary entry point for new claims, it needs to capture information necessary to set-up new claims and be able to pass the information to the LEGACY system. Once the information is passed, the LEGACY system owns all information contained in both systems, and it is

25   uneditable in the Claim Folder. However, the Claim Folder has more information than what is contained in the LEGACY system, and therefore allows certain information to be entered and modified once the claim is pushed to the LEGACY system.

The Claim Folder decomposes a claim into different levels that reflect the policy, the insured, the claim, the claimants, and the claimant's lines. Each level has a structured set of information that applies to it. For example, the claim level of the claim has information on the claim status, line of business, and performers. An individual line has information which includes the line type, jurisdiction, and property or vehicle damages. The claimant level contains contact information as well as injury descriptions.

The information at each level is grouped into sections for organization purposes. Each level has a details section that includes the basic information about the level.

The key levels on the Claim Folder and their information sections are:

- The Policy Level: Details and Covered Auto for auto claims, Covered Property for property claims and Covered Yacht for marine claims.
- The Claim Level: Details, Facts of Loss, Events, Liability. Liability is considered part of the Negotiation component and described there.
- The Participant Level: Details and Contact Information. For claimants, additional sections are shown to display, Events, Injury and Disability Management. The participant level is discussed in the Participant Component.
- The Line Level: Details, Damaged Vehicle for vehicle lines, Damaged Property for property lines, Damaged Yacht for marine lines, Events, Damages, and Negotiation. Damages and Negotiation are considered part of the Negotiation component and described there.

Events are triggered in the Claim Folder by performing certain actions like changing a jurisdiction, identifying an injury, or closing a line. Other general events are triggered in the Event Section on most levels by clicking the one that has occurred. These events are processed by the Event Processor and could

generate any number of responses. In one embodiment of the present invention, the primary response is to trigger new tasks in the Task Assistant for a claim.

User Interfaces

5

- Claim Folder UI
- Policy Level – Policy Details Tab
- Policy Level – Covered Vehicle Tab
- Policy Level – Covered Property Tab
10 - Policy Level – Covered Yacht Tab
- Claim level – Claim Details Tab
- Claim level – Facts of Loss Tab
- Claim level – Events Tab
- Claim level – Liability Tab
15 - Line level – Line Details Tab
- Line level – Damaged Property Tab
- Line level – Damaged Auto Tab
- Line level – Damaged Yacht Tab
- Line level – Events Tab
20 - Line level – Damages Tab
- Line level – Negotiation Tab
- Task Assistant
- File Notes
- Claim History
25 - Search Task Template
- Search for Correspondence
- Find Claims
- Version 7
- View File Folder
30 - Print Label

CLAIM FOLDER TREE AND MENU DESIGN

5

Claim Tree

The claim tree in the Claim Folder window decomposes the claim into policy, insured, claim, claimant, and line levels depending on the specific composition of the claim.

10

The policy level is always the first node in the claim tree and is identified by the policy number. Before the policy number is entered, the field is listed as "Unknown". If a claim is uncoded, the field is listed as "Uncoded". Selecting the policy level brings up the policy level tabs in the body of the Claim Folder.

15

The insured level is always the second node in the claim tree and is identified by the insured's name. Before the insured is identified, the field is listed as "Unknown". Selecting the insured level brings up the insured participant tabs in the body of the claim folder. Only one insured is listed at this level as identified

20 in the policy level tabs, however, multiple insureds can still be added. Additional insureds are shown in the participant list below the claim tree.

The claim level is always the third node in the claim tree and is identified by the claim number. When the claim level is selected, the claim level tabs appears in

25 the body of the Claim Folder.

After the claim level, all claimants are listed with their associated lines in a hierarchy format. When a claimant is added, a node is added to the tree, and the field identifying the claimant is listed as "Unknown". Once a participant has

30 been identified, partial or client, the name of the claimant is listed on the level.

When the level is selected, the participant level tabs for the claimant is shown in the body of the claim folder.

Line levels are identified by their line type. Before a line type is selected, the line
5   level is listed as "Unknown". When a line level is selected, the line level tabs for the specific line are shown in the body of the claim folder.

There are several things that can alter the claim tree once it has been set up. First, if a claimant or line is deleted, it is removed from the claim tree. A claim that is
10 marked in error does not change the appearance of the levels. Second, the claim, claimant, and line levels are identified by different icons depending on whether they are pushed to V7 or not. Third, when a line or claimant is offset, it is identified as such.

15 *Participant List*
The participant list box contains all the non-claimant and non-insured participants on the claim. (Claimants and insureds are shown in the claim tree and not repeated here.) Participants are shown with their name and role. When a participant is selected, the participant level tabs are displayed in the claim folder.
20

Claim Folder Menu Items
The claim folder menus contain the actions that a user would need to perform within the claim folder. They can all be accessed through keyboard selection. The menu options become enabled or disabled based on the state of the Claim
25 Folder. The Claim Folder can be in view mode or edit mode for a specific level in the Claim Tree. When the Claim Folder is in edit mode, most options are disabled until the user saves their changes and is returned to view mode. The enabling/disabling of menu options is also dependent on whether the claim or portions of the claim have been pushed to V7.

30

Claim Folder Tool Bar

The tool bar represents common action that a user performs that can be easily accessed by clicking the appropriate icon. There are five groups of button on the Claim Folder tool bar that represent, in order, common activities, adding new items to a claim, launching utilities, performing V7 activities, and accessing help functions. The enabling/disabling of tool bar buttons follows the same logic as for menu items.

Window Description

Control Name	Type	Description	Default Value/State
Claim Tree	Tree View	The Claim Tree lists the policy, insured, all of the claimants and their related lines in a claim tree format.	The current claim tree structure for the selected claim. The claim level is selected and the claim level tabs are displayed.
Participant List	List View	A list of all non-insured and non-claimant participants associated with a claim.	All participants who are not claimants or insureds for the claim and their roles
Edit Tool Bar Button	Command Button	Changes the tabs for the level selected in the claim tree or participant list view to edit mode.	Enabled when claim is in view mode.
Refresh Tool Bar Button	Command Button	Refreshes the current claim, including all Participant and Line information.	Enabled when claim is in view mode.
Find Tool Bar Button	Command Button	Opens the Claim Search window to allow the user to search for another claim	Enabled
Claim Allocation Tool Bar Button	Command Button	Opens the Claim Allocation window.	Enabled when claim is in view mode.

Control Name	Type	Description	Default Value/State
Manage Physical File Tool Bar Button	Command Button	Opens the Manage Physical File window.	Enabled when claim is in view mode.
Declare Event Tool Bar Button	Command Button	Opens the Declare Events window.	Enabled when claim is in view mode.
Claimant Tool Bar Button	Command Button	Adds claimant and opens Participant tabs in edit mode for entry of a new claimant level node	Enabled when claim is in view mode. V7 limit for claimants is 999, we will not edit this here.
Participant Tool Bar Button	Command Button	Adds a new participant and opens Participant tabs in edit mode.	Enabled when claim is in view mode.
Line Tool Bar Button	Command Button	Adds line and opens Line tabs in edit mode for entry of a new line level node.	Enabled when claim is in view mode and claimant context selected in claim tree. V7 limit for lines is 15 per claimant, this button will be disabled after 15 added.
Assign Performer Tool Bar Button	Command Button	Opens Assign Performer window	Enabled when claim is in view mode.
Print Screen Tool Bar Button	Command Button	Prints the current claim folder window.	Enabled
Task Assistant Tool Bar Button	Command Button	Launches Task Assistant for the current claim	Enabled when claim in view mode.
File Notes Tool Bar Button	Command Button	Launch File Notes for the current claim	Enabled when claim in view mode.
Claim History Tool Bar Button	Command Button	Launch Claim History for the current claim	Enabled when claim in view mode.
Correspondence Tool Bar Button	Command Button	Opens Forms and Correspondence window	Enabled when claim in view mode.
Push to V7 Tool Bar	Command Button	Open the terminal	Enabled when

Control Name	Type	Description	Default Value/State
Button		emulator window at the first V7 setup screen.	claim is in view mode and claim status is pre-push or open and there are new claimants or lines to push.
Make Payment Tool Bar Button	Command Button	Open the V7 PUEM screen in the terminal emulator window if a claimant or participant tied to one claimant is selected. Otherwise, display window that requires user to select a claimant.	Enabled when claim had been pushed to V7 and a participant is selected.
Help Tool Bar Button	Command Button	Opens Help	Enabled
Claim | Edit	Menu Option	Changes Claim tabs into Edit mode so that the user can make changes	Enabled when claim is in view mode.
Claim | Refresh	Menu Option	Refreshes the current claim, including all Participant and Line information.	Enabled when claim is in view mode.
Claim | Find	Menu Option	Opens the Claim Search window	Enabled
Claim | Save	Menu Option	Save the claim level when it is in edit mode.	Enabled when the claim level is in edit mode.
Claim | Claim Status | First Report Complete	Menu Option	Changes the status of the claim to "Unassigned" and creates First Report Complete Event.	Enabled when claim is in view mode and claim status is "New".
Claim | Claim Status | Assignment Complete	Menu Option	Changes the status of the claim to "Open" and creates Assignment Complete Event.	Enabled when claim is in view mode and claim status is "Unassigned".
Claim | Claim Status | Close	Menu Option	Initiates the close claim process	Enabled when claim is in view mode, V7 claim status is closed, and Millennium Claim Status is

Control Name	Type	Description	Default Value/State
			not "Closed" or "Archived"
Claim | Claim Status | Reopen	Menu Option	Changes the status of the claim to "Open".	Enabled when claim is in view mode and "Closed" or "Archived".
Claim | Claim Status | Mark In Error	Menu Option	Marks the current claim and all of its lines in error. Expires all participants.	Enabled when claim is in view mode, and not pushed to V7.
Claim | Allocate	Menu Option	Opens the Claim Allocation window.	Enabled when claim is in view mode.
Claim | Manage Physical File	Menu Option	Opens Physical File window	Enabled when claim is in view mode.
Claim | Declare Event	Menu Option	Opens Declare Event window	Enabled when claim is in view mode.
Claim | Close Claim Folder	Menu Option	Closes current claim folder window	Enabled
Edit | Cut	Menu Option	Move selected text to the clipboard	Disabled
Edit | Copy	Menu Option	Copy selected text to the clipboard	Disabled
Edit | Paste	Menu Option	Paste text from the clipboard	Disabled
View | Collapse All	Menu Option	Collapses the claim tree	Enabled
View | Expand All	Menu Option	Expand the claim tree	Enabled
Policy | Edit	Menu Option	Opens policy tabs in edit mode.	Enabled when claim is in view mode.
Policy | Save	Menu Option	Save current policy tab information.	Enabled when policy level is in edit mode.
Participant | New | Claimant	Menu Option	Opens Participant tabs in edit mode for entry of a new claimant level node in the claim tree.	Enabled when claim in view mode.
Participant | New | Insured	Menu Option	Opens Participant tabs in edit mode for entry of a new insured level node in the claim tree.	Enabled when claim in view mode.

Control Name	Type	Description	Default Value/State
Participant | New | Other	Menu Option	Opens Participant tabs in edit mode for entry of a new entry in the Participant list.	Enabled when claim in view mode.
Participant | Edit	Menu Option	Puts currently selected participant tabs into edit mode.	Enabled when claim is in view mode and participant selected in tree or list box.
Participant | Save	Menu Option	Saves information changed on participant tabs and returns claim to view mode.	Enabled only when a participant level is in edit mode.
Participant | Delete	Menu Option	Deletes selected participant	Enabled only when claim is in view mode and participant is selected.
Line | New	Menu Option	Adds new line to claim tree and opens line tabs in edit mode.	Enabled when claim is in view mode, claimant has been selected, and limit of 15 lines per claimant has not been exceeded.
Line | Edit	Menu Option	Puts Line tabs into edit mode so that the user can change line details	Enabled when claim is in view mode and line is selected.
Line | Save	Menu Option	Save information entered on line tabs and returns claim to view mode.	Enabled when a line is in edit mode.
Line | Change Status | Close	Menu Option	Changes status of a line in the claim folder to "Closed"	Enabled when claim is in view mode, a line is selected, the line is not closed, and its V7 status is closed.
Line | Change Status | Reopen	Menu Option	Changes the status of the line selected to	Enabled when claim is in view

Control Name	Type	Description	Default Value/State
		"Open".	mode, a line is selected, and line is "Closed".
Line | Change Status | Mark in Error	Menu Option	Marks selected line in error.	Enabled when claim is in view mode, a line is selected, and line has not been pushed.
Line | Allocate	Menu Option	Opens the Claim Allocation window.	Enabled
Performers | Assign	Menu Option	Opens the Assign Performers window	Enabled when claim is in view mode.
Performers | View All	Menu Option	Displays all claim performers assigned to the claim in View Performer UI.	Enabled when claim is in view mode.
Utilities | Print Screen	Menu Option	Prints current screen.	Enabled
Utilities | View Task Assistant	Menu Option	Opens Task Assistant window for current claim.	Enabled when claim is in view mode.
Utilities | Create New File Note	Menu Option	Opens File Notes window for current claim.	Enabled when claim is in view mode.
Utilities | View Claim History	Menu Option	Opens Claim History window for current claim.	Enabled when claim is in view mode.
Utilities | Create Correspondence	Menu Option	Opens Forms and Correspondence window.	Enabled when claim is in view mode.
Version 7 | Push Claim	Menu Option	Launches V7 to start the push process.	Enabled when claim is in view mode and in "Pre-Push" status or open when there are unpushed claimants and lines.
Version 7 | Undo Push	Menu Option	Reverts claim to pre-push status.	Enabled when claim is in view mode and status is "Push-Pending".

Control Name	Type	Description	Default Value/State
Version 7 | Make Payment	Menu Option	Open the V7 PUEM screen in the terminal emulator window if a claimant or participant tied to one claimant is selected. Otherwise, display window that requires user to select a claimant.	Enabled when claim had been pushed to V7 and a participant is selected.
Help | Contents	Menu Option	Opens help file to content menu.	Enabled
Help | Search For Help On	Menu Option	Open help file to search window.	Enabled
Help | About	Menu Option	Opens window displaying information about the application.	Enabled

Window Details

Control Name	Initial Focus	Default Button	Tab Order
Claim Tree	Yes		1
Participant List			2
Claim Menu			3
Edit Menu			4
View Menu			5
Policy Menu			6
Participant Menu			7
Line Menu			8
Performer Menu			9
Utilities Menu			10
Version 7 Menu			11
Help Menu			12

5 **CAR Diagram**

Control Name	Action	Response	Short Cut Key	Mnemonic Key
Claim Tree	Click	• Highlights Node in Tree • Disable participant in list view if one selected previously • Shows related tabs in		

Control Name	Action	Response	Short Cut Key	Mnemonic Key
		view mode.		
		• Enable appropriate menu items and tool bar buttons.		
	Double Click	• Level selected in tree enters Edit mode.		
All Text Fields	Highlight	• Enable Cut and Copy.		
Participant List	Click	• Highlights participant in list box		
		• Deselects level in claim tree if one selected previously		
		• Shows related tabs in view mode.		
		• Enable appropriate menu items and tool bar buttons.		
	Double Click	• Participant selected in list view enters Edit mode.		
Edit Tool Bar Button	Click	• Changes the tabs for the level selected in the claim tree or participant list view to edit mode.		
Refresh Tool Bar Button	Click	• Refreshes the current claim, including all Participant and Line information.		
Find Tool Bar Button	Click	• Opens the Claim Search window to allow the user to search for another claim		
Claim Allocation Tool Bar Button	Click	• Opens the Claim Allocation window.		
Manage Physical File Tool Bar Button	Click	• Opens the Manage Physical File window.		
Declare Event Tool Bar Button	Click	• Opens the Declare Events window.		
Claimant Tool Bar Button	Click	• Adds claimant and opens Participant tabs in edit mode for entry of a new claimant level		

Control Name	Action	Response	Short Cut Key	Mnemonic Key
		node		
Participant Tool Bar Button	Click	• Adds new participant and opens Participant tabs in edit mode.		
Line Tool Bar Button	Click	• Adds line and opens Line tabs in edit mode for entry of a new line level node.		
Assign Performer Tool Bar Button	Click	• Opens Assign Performer window		
Print Screen Tool Bar Button	Click	• Prints the current claim folder window.		
Task Assistant Tool Bar Button	Click	• Launches Task Assistant for the current claim		
File Notes Tool Bar Button	Click	• Launch File Notes for the current claim		
Claim History Tool Bar Button	Click	• Launch Claim History for the current claim		
Correspondence Tool Bar Button	Click	• Opens Forms and Correspondence window		
Push to V7 Tool Bar Button	Click	• Open the terminal emulator window at the first V7 setup screen.		
Make Payment Tool Bar Button	Click	• Open the V7 PUEM screen in the terminal emulator window if a claimant or participant tied to one claimant is selected. Otherwise, display window that requires user to select a claimant.		
Help Tool Bar Button	Click	• Opens Help		
Claim | Edit	Click	• Changes Claim tabs into Edit mode so that the user can make changes		
Claim | Refresh	Click	• Refreshes the current claim, including all Participant and Line	Ctrl+R	

Control Name	Action	Response	Short Cut Key	Mnemonic Key
		information.		
Claim | Find	Click	• Opens the Claim Search window		Ctrl+F
Claim | Save	Click	• Save the claim level when it is in edit mode.		
Claim | Claim Status | First Report Complete	Click	• Changes the status of the claim to "Unassigned" and creates First Report Complete Event.		
Claim | Claim Status | Assignment Complete	Click	• Changes the status of the claim to "Open" and creates Assignment Complete Event.		
Claim | Claim Status | Close	Click	• Initiates the close claim process		
Claim | Claim Status | Reopen	Click	• Changes the status of the claim to "Open".		
Claim | Claim Status | Mark In Error	Click	• Marks the current claim and all of its lines in error. Expires all participants.		
Claim | Allocate	Click	• Opens the Claim Allocation window.		
Claim | Manage Physical File	Click	• Opens Physical File window		
Claim | Declare Event	Click	• Opens Declare Event window		
Claim | Close Claim Folder	Click	• Closes current claim folder window		
Edit | Cut	Click	• Move selected text to the clipboard		Ctrl + X
Edit | Copy	Click	• Copy selected text to the clipboard		Ctrl + C
Edit | Paste	Click	• Paste text from the clipboard		Ctrl + V
View | Collapse All	Click	• Collapses the claim tree		
View | Expand All	Click	• Expand the claim tree		
Policy | Edit	Click	• Opens Policy tabs in edit mode		
Policy | Save	Click	• Save policy information and returns		

Control Name	Action	Response	Short Cut Key	Mnemonic Key
		tabs to view mode.		
Participant | New | Claimant	Click	• Opens Participant tabs in edit mode for entry of a new claimant level node in the claim tree.		
Participant | New | Insured	Click	• Opens Participant tabs in edit mode for entry of a new insured level node in the claim tree.		
Participant | New | Other	Click	• Opens Participant tabs in edit mode for entry of a new entry in the Participant list.		
Participant | Edit	Click	• Puts currently selected participant tabs into edit mode.		
Participant | Save	Click	• Saves information changed on participant tabs and returns claim to view mode.		
Participant | Delete	Click	• Deletes selected participant		
Line | New	Click	• Adds new line to claim tree and opens line tabs in edit mode.		
Line | Edit	Click	• Puts Line tabs into edit mode so that the user can change line details		
Line | Save	Click	• Save information entered on line tabs and returns claim to view mode.		
Line | Change Status | Close	Click	• Changes status of a line in the claim folder to "Closed"		
Line | Change Status | Reopen	Click	• Changes the status of the line selected to "Open".		
Line | Change Status | Mark in Error	Click	• Marks selected line in error.		
Line | Allocate	Click	• Opens the Claim Allocation window.		

Control Name	Action	Response	Short Cut Key	Mnemonic Key
Performers | Assign	Click	• Opens the Assign Performers window		
Performers | View All	Click	• Displays all claim performers assigned to the claim in View Performer UI.		
Utilities | Print Screen	Click	• Prints current screen.		Ctrl+P
Utilities | View Task Assistant	Click	• Opens Task Assistant window for current claim.		
Utilities | Create New File Note	Click	• Opens File Notes window for current claim.		
Utilities | View Claim History	Click	• Opens Claim History window for current claim.		
Utilities | Create Correspondence	Click	• Opens Forms and Correspondence window.		
Version 7 | Push Claim	Click	• Launches V7 to start the push process.		
Version 7 | Undo Push	Click	• Reverts claim to pre-push status.		
Version 7 | Make Payment	Click	• Open the V7 PUEM screen in the terminal emulator window if a claimant or participant tied to one claimant is selected. Otherwise, display window that requires user to select a claimant.		
Help | Contents	Click	• Opens help file to content menu.		
Help | Search For Help On	Click	• Open help file to search window.		
Help | About	Click	• Opens window displaying information about the application.		

Data Elements

Literal	Length	Control Type	Data Entity Tie	Edit Rules*	Error Handling
Claim Tree		Tree View			
- Policy		Tree View Node	Policy Number (Policy)		
- Insured		Tree View Node	Participant Preferred Name (Insurance Involvement)		
- Claim		Tree View Node	Claim Number (Claim)		
- Claimant		Tree View Node	Particiapant Preferred Name (Insurance Involvement)		
- Line		Tree View	Line Type (Line)		
Participant List Box		List View	Participant Preferred Name and Role (Insurance Involvement & Involvement Role)		

5

Commit Points

10
Claim		Save Menu Option – Saves all claim level data
Policy		Save Menu Option – Saves all policy level data
Participant		Save Menu Option – Saves all participant level data
Line		Save Menu Option – Saves all line level data
Claim		Close Claim Folder Menu Option – Prompts user to save changes

if in edit mode.

CLAIM HISTORY

5 *Definition*

Claim history shows information in one user interface that is intended to include all the constituent elements of a claim file. The four types of history included in the component are searchable by common indexing criteria like participant, performer, and claim phase. A caption report can be produced which shows the

10 history selected in a document format.

Value

Claim history provides the users with one common interface through which to view a large variety of information about the claim. It includes all history available on a claim, and is expanded as claim capabilities are built, like

15 incoming mail capture. Users develop customized views of history based on any criteria the history can be indexed by, and these reports are saved as customizable Word documents. The way the history information is indexed provides quick access to pertinent data needed to respond to a variety of requests.

Key Users

20 All members of the claims organization can use claim history as a way to quickly see all activity performed on a claim. This utility increases the ability to locate key information regarding any claim.

Component Functionality

Claim history is a component that contains a simple process to retrieve history

25 from the other components in the system. It contains no native data itself. Even viewing a history element is done in the component window where the item was first captured.

The second key process of claim history is to produce a caption report of all history elements according to the items the user wants to include.

There are two user interfaces needed for this component that correspond to the two key functions above:

- Claim History Search: This window utilizes the claim phase, participant, performer and history type fields on each history record to help the user narrow the search for specific history.
- Caption Report: This report uses the functionality of Word to produce a report of each history item the user wants to see and its associated detail. Since the report is produced in Word, it can be fully customized according to many different needs.

User Interfaces

- Claim History Search
- Caption Report (Word document, not UI design)

FORMS AND CORRESPONDENCE

Definition

The Forms & Correspondence component supports internal and external Claim communication and documentation across all parts of the claims handling process.

The Forms and Correspondence - Create Correspondence function provides the ability to search for a template using various search criteria, select a template for use and then leverage claim data into the selected template.

The Forms and Correspondence - Template Maintenance function is a tool for the librarian to create, delete, and update Correspondence templates and their associated criteria.

5 Some specific processes supported by Forms & Correspondence are:
- Reporting of claims
  - to state/federal agencies, etc. at First Notice of Loss
  - internal requests for information
- Advising Participants
10 - Contacting Participants
- Performing Calculations
- Creating correspondence for claims or non-claims

Value

15

The Forms and Correspondence component supports user in creating documentation.

Leveraging information from the claim directly into correspondence reduces the
20 amount of typing and dictating done to create forms and letters. The typical data available to the templates should include: author, addressee, claim number, date of loss, insured name, policy number, etc. A librarian adds and maintains standardized forms and letters in logical groupings made available for the entire company.

25

Key Users

Claim employees are the primary users of the Forms and Correspondence component, but it can be used by anyone who has access to the system to create
30 documents using existing templates.

Forms and Correspondence librarians use the system to create, update or remove templates.

Component Functionality

Forms and Correspondence - Create Correspondence

5 1. Search for a template based on search criteria.

2. Create a correspondence from a template using claim data.

3. Create a correspondence from a template without using claim data.

4. View the criteria for a selected template.

5. View the Microsoft Word template before leveraging any data.

10

Forms and Correspondence - Template Maintenance

1. Search for a template based on search criteria.

2. Create, duplicate, edit, and delete Correspondence templates and their criteria.

3. Internally test and approve newly created/edited templates.

15 4. Properly copy Word templates for NAN distribution.

User Interfaces

- Search for Correspondence
- Correspondence Details
20 - Associate Fields
- Maintain Correspondence Search
- Correspondence Template Information – Details tab
- Correspondence Template Information – Criteria tab
- Microsoft Word

25

FILE NOTES

Definition

File notes captures the textual information that cannot be gathered in discrete data elements as part of claim data capture. They are primarily a documentation tool,

5 but also are used for internal communication between claim professionals. Users can sort the notes by participant or claim phase (medical, investigation, coverage, etc.) in order to permit rapid retrieval and organization of this textual information.

Value

10 File notes speeds the retrieval and reporting of claim information. A file notes search utility with multiple indexing criteria provides claim professionals and supervisors with the ability to quickly find a file note written about a particular person or topic. The file notes tool utilizes modern word processing capabilities which speed entry, reduce error, and allow for important information to be

15 highlighted. Furthermore, the categorization and key field search eases the process of finding and grouping file notes. Finally, file notes improves communication as they can be sent back and forth between those involved in managing the claim.

Key Users

20 All members of the claims organization can utilize file notes. External parties via RMS can view file notes marked General. This utility increases the ability to locate key information regarding a claim. Anyone who wants to learn more about a claim or wants to record information about a claim utilizes the file notes tool.

25 *Component Functionality*

File Notes searching is included as part of the claim history component which allows the user to search the historical elements of a claim file including tasks, letters, and significant claim change events.

The user interfaces that are needed for this component are:

- The File Notes Search (part of Claims History component): This window utilizes the claim phase fields on the file notes record to help the user narrow the search for specific file notes. Also, it allows users to view all file notes that meet specified criteria in a report style format.

- File Notes Entry: The window used to record the file note. It embeds a word processing system and provides the ability to categorize, indicate a note as company (private) vs. general (public), save the note as a draft or a final copy, and send the note to another person.

User Interfaces

- File Notes
- Draft File Note Review
- Participant Search
- Performer Search

ADDRESS BOOK

Definition

Address Book is the interface between the claims system and the Client database. The Client application is a new component designed to keep track of people or organizations that interact with RELIANCE for any reason, but claims are most likely the first application to use Client. The Address Book is accessed directly from the Desktop and from the Claim Folder.

The Address Book meets several needs within the claim organization. Although, its primary function is to support the adding of participants to a claim, it acts as a

pathway to the Client database for searching out existing participants, and adding new people or organizations to the corporate database.

The Client database maintains information on names, addresses, phone numbers,
5   and other information that always applies to a person or organization no matter what role they play on a claim.

Value

Address Book provides a common definition of people or organizations that
10   interact with RELIANCE, and therefore provides a much more efficient means of capturing this information. Each Client database entry provides the ability to link a person or organization to all the different roles that they play across the organization, and therefore makes retrieving information on a client by client basis quick and easy.

15

There are many benefits to RELIANCE by having a common address book. Information on people and organizations is leveraged into other activities like enabled tasks that lookup a client's phone numbers when a call needs to be made. Information that has been redundantly stored in the past can be entered once and
20   reused. Once all areas of RELIANCE use the Client application, different areas of the company can share definitions of individuals and organizations.

Component Functionality

Address Book allows users to add, edit and delete records from the Client
25   database. It also provides a robust search facility, including phonetic name searches to find people contained in the Client database.

There are two primary user interfaces for the Address Book:

- Find Address Book Entry – This is a search window that allows a user to find records in the Client database using names, addresses, phone numbers, and other identifiers. From this window, specific records can be selected and attached as participants on claims.

5
- Maintain Address Book Entry – This window allows users to add or edit information about a client by specifying their names, addresses, phone numbers, email information, and identification numbers like a SSN or TIN.

10   The Address Book is created concurrently with the Client application to make sure that a consistent design approach is followed.

Key Users

All members of the claim organization use the Address Book to look up
15   information on people and organizations in the client database. Those who set up and handle claims use the Address Book to identify participants.

User Interfaces

- Find Client
20
- Maintain Client

INDEX

Definition

25
The Index, or Claim Search, component provides the ability to locate claims within the system using various search criteria. The criteria cover a wider variety of search capabilities than exist today including, but not limited to, claim performers, participants, phonetic name searches, addresses, roles, offices, and

lines of business. The search results display selected claim, participant, and performer data to help identify each claim.

The Index component also allows easy navigation to various claim components
5   like the Claim Folder, once a claim has been identified. It can be accessed from the Desktop and from any open Claim Folder.

The Index component is designed to support several business processes within the claim organization. Its functions are critical to improving claim staff
10 productivity and customer service in the following areas:

- Matching Mail

    The capabilities of the Index search make it easier to identify the claim a piece of mail belongs to based on criteria used to identify
15 claims in forms, correspondence, and bills. The performers for a claim can also be identified for mail routing purposes.

- Phone Inquiries

    This window is the primary point to handle incoming phone inquiries for any claim. Users can find claims quickly without having to burden
20 the caller with requests for additional information.

- Duplicate Claims

    Prior to setting up new claims, checks can be done to ensure that the claim has not already been entered into the system. The additional search capabilities provide a greater assurance that duplicate claims
25 will not be entered. This reduces the need to delete or merge claim records.

- Fraud Identification

    Because claims can be searched easily by participant and other criteria, fraud questions can be easily researched. This is not the
30 primary purpose of this component, however.

Value

Index reduces the time required to find existing claims, and also reduces potential rework from not finding claims when they are needed for matching mail or

5   duplicate checks.

Key Users

Claim employees are the primary users of the Index window, but it can be used by anyone who has access to the system to access claims without having to

10   memorize tracking numbers.

Component Functionality

Index is primarily a robust search engine that quickly and efficiently searches for claims. It is not a component that stores its own data, as it is primarily focused

15   on pointing users more quickly and directly to claim data.

Index is composed of one search window that follows the format of all other search windows in the system.

20   *User Interfaces*

- Find Claims

25   **INJURY**

Definition

The Injury component captures versions of a claimant's injuries as they progress. This window captures injury information in the form of discrete data fields, reducing the need for free form text file notes. Capturing data, instead of text,

allows the injury to be closely tracked and quickly reported. The data can also serve as feedback statistics, i.e. for building best claims practices and in risk selection. The preferred method of identifying and documenting injuries is the ICD-9 code. The user can enter or search for the ICD-9 code using descriptors or

5   numbers.

Value

Data on every injury is captured and summarized in a consistent, accessible

10   format, making recording and reviewing the case considerably less time consuming and more organized, allowing the adjuster to focus on desired outcomes. This "snapshot" of the current status and history of an injury greatly facilitates handing off or file transfers between claim professionals. Additionally, the discrete data field capture enables the use of events to identify action points in

15   the lifecycle of a claim that has injuries.

Key Users

All members of the claims organization can utilize the Injury component. This component increases the ability to locate and summarize key information

20   regarding an injury.

Component Functionality

Injury is an aspect of participant information, which is related to the claimant participants on the claim. The participant component relates clients to all other

25   claim-related entities. Information on injuries will be related to participant records and displayed at the participant level information in the Claim Folder. New entities are needed to implement injury data capture: injury and ICD-9 search. The Injury component interacts with five other components: Claim Folder-which contains Disability Management data about a claimant; Participant-

which lists the individuals associated with the claim; as well as File Notes, Task Assistant and the Event Processor. The injury component also uses Microsoft WORD to create a formatted, historical injury report for a particular individual.

5 The user interfaces that are needed for this component are:

- Injury: This is the primary injury window which captures basic injury report data, including: the source of the injury report, the date of the injury report, a Prior Medical History indicator, and then a detailed list
10 of the injuries associated with that report. The detailed list includes discrete fields for the following data: ICD-9 code, body part, type, kind, severity, treatment, diagnostic, a free form text description field, and a causal relation indicator.
- ICD-9: This is the search window for locating ICD-9 codes and
15 associated descriptions.
- Disability Management: This window contains a subset of participant data fields that enables more effective injury management.

User Interfaces

20

- Claim Folder – Participant Level - Injury Tab
- ICD-9 Search Window
- Claim Folder – Participant Level - Disability Management Tab

25

NEGOTIATION

Definition

Figure **10** is an illustration of the Negotiation component of one embodiment of
30 the present invention. Negotiation provides a single, structured template that is

supplemented by supporting views, to capture events regarding a negotiation. The negotiation interface **1000** captures key elements of a negotiation, such as a settlement target range, current demands and offers, and Supporting Strengths and Opposing Assertions of the claim. Negotiation information is gathered in

5   discrete data elements **1002**, enabling the capability to generate events **1006** based on key attributes or changes in a negotiation. These events **1006** are then sent to a common event queue **1008**. The negotiation component **1000** interfaces with the File Notes **1004** component to provide additional documentation capability, in a non-structured format. The negotiation template is supported by

10   all other data contained in the Claim Folder.

Value

Data on every case is summarized in a consistent, accessible format, making

15   recording and reviewing the case considerably less time consuming and more organized, allowing the adjuster to focus on negotiation strategy and desired outcomes. This "snapshot" of the current status greatly facilitates handing off or file transfers between claim professionals. Additionally, the discrete data field capture enables the use of events to identify action points in a negotiation.

20

Key Users

All members of the claims organization can utilize Negotiation. This component increases the ability to locate and summarize key information regarding a

25   negotiation.

Component Functionality

Negotiation is a type of resolution activity, which is part of the claim component

30   of the claims entity model. The claim component is the central focus of the

claims entity model, because it contains the essential information about a claim. The claim component supports the core claim data capture functionality, first notice processes, and resolution activity for claims. The main types/classes of data within the claim component are: Claim, Claimant, Line, Claim History,

5   Resolution Activity, Reserve Item, and Reserve Item Change. Three entities are needed to implement negotiation: resolution activity, claim and claim history. There is also interaction between the Negotiation component and the Task Assistant, File Notes and Event Processor components.

10   The user interfaces needed for negotiation are:

- Negotiation: This window captures demand and offer data, including: amount, date, type and mode of communication. The target settlement range, lowest and highest, is captured, along with strengths and
15   weaknesses of the case.

Supporting user interfaces, which are also part of the Claim Folder, include:

- Liability (claim level tab): This window is used to document liability
20   factors in evaluating and pricing a claim. The liability factors include percent of liability for all involved parties; form of negligence that prevails for that jurisdiction; theories of liability that the claim handler believes to be applicable to the claim. Used prior to developing negotiation strategy.
25   - Damages (line level tab): This window provides the capability for pricing and evaluating a claim based on incurred and expected damages. Used prior to developing negotiation strategy.

User Interfaces

30

- Claim Folder – Line Level - Negotiation Tab
- Claim Folder – Claim Level - Liability Tab
- Claim Folder – Line Level - Damages Tab

<div style="text-align:center">5   **ORGANIZATION**</div>

Definition

Figure **11** is a flow diagram of the operations utilized by the Organization
component in accordance with one embodiment of the present invention. The
Organization component **1100** allows common information for the people who
10   perform work on claims to be stored, searched, and reused across all the claims
they work.

In one embodiment of the organization component **1100**, all employee records
are kept in a common database **1102** so that they can be attached to the specific
15   claims they work, located in a claim database **1104**. The common information
that is kept on the employee record includes name, location, phone, and some
minimal organizational context information like office or division. This is the
minimum required to support the tracking of performers on claims. The employee
information **1102** is then linked **1106** to the claim information **1104** and the
20   databases are updated **1108**. Having linked the employees **1102** with the claims
1104 they are working on, the database can be searched by employee or claim
1110.

However, this version of the organization can be expanded to include
25   organization relationships (specifically tracking where an employee falls in the
organization structure), groups of individuals as performers for claim assignment,
and claim allocation within the organization structure. These capabilities are to
support any notion of caseload analysis, management reporting, or automated
assignment that would need to be included.

30

Value

By tracking common definitions of employees across claims, indexing capabilities are improved and performers on claims are accurately tracked.

5 ***Key Users***

The primary users of the organization capabilities are the administrative personnel who set up performers, as well as the technicians who track who is working a claim.

10 ***Component Functionality***

The design of the minimum scope of the organization component includes a search window to find employees in the organization and a detail window to see specific information on each employee.

User Interfaces

15

- Organization Entity Search
- Add/Edit Organization Entity

20 **Participant**

Definition

Figure **12** is an illustration of the Participant component in accordance with one embodiment of the present invention. Participant **1200** provides the link between

25 claims and individuals and organizations stored in the Client database and accessed through the Address Book **1202**. Participant links clients to claims **1204** by defining the roles that they play, e.g. claimant, driver, or doctor. It reuses the

information contained in the Address Book **1202** so that it does not have to be reentered for each participant.

The participant component also allows linkages **1206** to be made between participant and to various items on claims. A doctor can be linked to the claimant they treat and a driver can be linked to the damaged vehicle they were driving.

Once a participant has been added to a claim, additional information **1208** that is specific to that claim can be attached. This information includes injury, employment, and many other types of information that are specific to the role that a person or organization plays in a claim.

The business processes primarily supported by Participant **1200** are:

- Recording Involvement in a Claim

  There is a basic data capture requirement to keep track of individuals and organizations involved in a claim, and this is done most efficiently using the participant approach.

- Recording Role Specific Information

  Address Book **1202** stores information that can be reused across claims, but the Participant component **1200** needs to maintain the information that is specific to an individual or organization's involvement in a specific claim.

- Making Contact with Clients

  Because participant ties back to the common Address Book **1202**, any contact information contained there can be quickly and easily obtained.

- Forms and Correspondence **1210**

  Leveraging address information into letters provides an efficiency enablement to all users who don't need to look up name and address information.

- Categorizing History Information

  Participants are used to categorize history items like tasks and file notes so that information relating to a single participant on a claim can be easily retrieved.

5
- Claim Indexing

  Attaching participants to a claim allows the Index component to be more effective in the processing of claim inquires.

Key Users

10 The primary users of the Participant components **1200** are those who work directly on processing claims. They are the ones who maintain the participant relationships.

Claims professionals who deal with injuries use the Participant tabs in the claim
15 folder to track injuries and manage disabilities for a better result on the claim.

Value

Because the Participant component **1200** only seeks to define the roles that individuals and organization play across all claims, there is no redundant entry of
20 name, address, and phone information. This is all stored in the Address Book **1202**.

The number of potential participant roles that can be defined is virtually limitless, and therefore expandable, as the involvement of additional people and
25 organizations needs to be captured.

Component Functionality

Most participant functionality is executed within the context of the Claim Folder.
The Claim Folder contains participants levels in two ways. First, claimants are
shown in the claim tree on the left-hand side of the window. Below this, other
participants are shown in a list. Selecting any participant displays a set of
5   participant information tabs that displays the following information:

- Participant Details – Basic information about the role that a
  participant plays in a claim and all the other participants that are
  associated to it.
10   - Contact Information – Information from the Address Book on names,
  addresses, and phone numbers.
- Injury – Specific information on the nature of injuries suffered by
  injured claimants.
- Disability Management – Information on injured claimants with
15   disabilities.

Only the first two tabs will be consistently displayed for all participants. Other
tabs can appear based on the role and characteristics of a participant's
involvement in a claim.
20

Adding or editing participant role information is actually done through the
Address Book **1202** search window. The process is as simple as finding the
Address Book **1202** record for the intended participant and specifying the role the
participant plays in the claim. Once this is done, the participant will be shown in
25   the Claim Folder, and additional information can be added.

The notion of a participant is a generic concept that is not specific to claims
alone. It is a based on design pattern that can be expanded as additional claims
capabilities are built. Any involvement of an individual or an organization can be
30   modeled this way.

User Interfaces

- Participant Level – Participant Details Tab
- Participant Level – Contact Information Tab
- Participant Level – Events Tab
- Participant Level – Injury Tab (Injury Component)
- Participant Level – Disability Management Tab (Injury Component)
- View Participant List

PERFORMER

Definition

The Perforer component allows organizational entities (individuals, groups, offices, etc.) to be assigned to various roles in handling the claim from report to resolution. The Performer component is utilized on a claim-by-claim basis.

A performer is defined as any individual or group that can be assigned to fulfill a role on a claim.

The Performer component supports the assignment processes within the claim handling process. This goes beyond the assignment of claim at FNOL. This component allows the assignment of work (tasks) as well.

Some specific processes supported by Performer are:

- Assign claims

  identification of different roles on the claims in order to assign the claim (Initiate Claim - DC Process work)
- Keeps roles and relationships of performers within claims
- Assigning tasks

- Reassignments
- Supports Initiate claim process - assignment
- Search mechanism for employees, offices
- All performers should be in the Organization component
5 - Provides history of assignments

Value

The Performer component allows the assignment of roles or tasks to individuals or groups. The data about performers resides in a common repository: the Organization component.

10

The Performer component reduces the time required to find employees, teams or any potential performer, and ensures consistency of data.

Key Users

15   The primary users of the Performer component are those who work directly on processing claims. They are the ones who maintain the assignment of roles or tasks related to a claim.

Component Functionality

The Performer component supports an informational function and an assignment
20   function.

1. View details for performers (employee, office, unit, etc.). These details may suggest organizational entity relationships but in no way define or maintain them.

2. View all performers assigned to a claim, currently and historically (includes
25   individuals, groups, offices, etc.)

3. Assign performers to a claim - at the claim level, claimant, and supplement levels (including individuals, office, groups, etc.)

User Interfaces

- Assign Performer
- Performer Roles
- View Performer List

5

TASK ASSISTANT

Definition

The Task Assistant is the cornerstone of a claim professional's working
environment. It provides diary functions at a work step level that allow the
10 management of complex claim events. It enables the consistent execution of
claim best practices by assembling and re-assembling all of the tasks that need to
be performed for a claim based on detailed claim characteristics. These
characteristics come from regulatory compliance requirements, account servicing
commitments, and best practices for handling all types of claims. The Task
15 Assistant also provides mechanisms that automate a portion of or all of the work
in performing a task to assist the claim professional in completing his or her
work. Once a task is completed, the Task Assistant generates a historical record
to document the claim handler's actions.

20 The Task Assistant is...

- A method for ensuring consistent execution of regulatory requirements,
account servicing commitments and claim handling best practices
- A source of automated assistance for claim professionals
- An organization-wide communication tool within the context of a claim (it
25 does not replace Lotus Notes).
- A mechanism for making claims strategy common practice and sharing
corporate experience
- A diary application to keep track of claims
- A historical tracking tool

- A way to get a claim professional's or a team leader's attention
- A mechanism for making process changes in the organization quickly

Within the Task Assistant, claim professionals have the ultimate control to
5   determine if and when tasks need to be completed. They also have the ability to
add tasks to the list to represent work they do that is not reflected in standard
definitions of tasks in the system. This supports a vision of the claim
professional as a knowledgeable worker who spends most of his or her time
focused on a successful result through investigation, evaluation, and negotiation
10 of the best possible outcome.

Value

The Task Assistant reduces the time required to handle a claim by providing the
claim professional with the automatic scheduling of claim activity. It helps the
15 claim professional remember, perform and record tasks completed for every
claim. Completed tasks are self-documenting and remain part of the claim
history.

The Task Assistant also ensures the consistent handling of claims throughout the
20 organization, and by doing so can significantly impact expenses and loss costs.
Furthermore, it helps ensure regulatory compliance and the fulfillment of account
promises. It supports the teamwork required in handling difficult claims as a
structure communication mechanism.

25 The automated enablements for tasks reduce the amount of time claim
professionals have to spend on low value-added activities such as writing
correspondence. They can therefore spend a larger amount of time investigating,
evaluating, and negotiating each claim.

Key Users

While claim professionals are the primary users of the Task Assistant, others use the application as well. The entire claims department utilizes the Task Assistant to structure work and communicate with one another. Team leaders use the Task

5   Assistant to conduct file review and to guide the work of the claim professional. Administrative staff use the Task Assistant as a means to receive work and to communicate the completion of that work. Claim professionals use the Task Assistant to complete work and to request assistance from team leaders and specialty claim professionals.

10

The Task Assistant requires a new type of user to set-up and maintain the variety of tasks that are created. A task librarian maintains the task library, which contains the list of all the standardized tasks across the organization. The librarian defines rules which cause tasks to be placed on task lists based on claim

15   characteristics, dates which define when tasks are due, and task enablement through other applications.

Component Functionality

Figure **13** is a flow diagram of the operations utilized by the Task Assistant

20   component of the present invention. The processing of tasks through the Task Assistant comprises the lifecycle of the task from its creation to its completion or deletion. In first operation **1300**, the Task engine provides tasks to the Task Assistant. In the second operation **1302**, the Task Assistant then displays the list of tasks provided by the Task Engine. In the third operation **1304**, the user is

25   allowed to add tasks and edit tasks provided by the Task Engine. The fourth operation **1306** occurs as the claim is processed. As the claim is processed, the user and the Task Engine determine when the various tasks are completed. When a tasks is completed, the fifth operation **1308** occurs. In the fifth **1308** operation, a historical record is generated for any tasks which is determined to be

30   completed.

The key user interfaces for this component are:

5

- The Task Assistant: This is the utility that supports the population, execution, and historical tracking of tasks. It allows users to perform tasks, complete tasks, and remove tasks that have been automatically added.

- The Task Workplan: This user interface allows the user to strategize the plan for a specific claim. It shows tasks attached to their

10

respective levels of the claim including lines, participants, and the claim itself.

- Task Enablement Windows: There are many windows that can be added to enable task with other applications such as telephone support, forms and correspondence, and file notes. The number of

15

potential task enablements is virtually limitless.

- Task Entry: Allows a user to add new task that weren't automatically added to the task list to cover situations where the claim handler wants to indicate work to be done that is not reflected by the standard task definitions in the task library.

20

Behind the functioning of the Task Assistant, the Task Engine continually evaluates messages sent from other components and determines based on the rules established by the task librarian, which tasks should be populated on the Task Assistant. Messages are sent to the Task Assistant when something

25 significant occurs in another component. The messages contain the characteristics the Task Engine needs to evaluate in order to place the proper tasks on the task list.

User Interfaces

30

- Task Assistant

- Reassign Task
- Edit/Add Task
- Clear Task
- Mark Task In Error
- Build Workplan
- Participant Search
- Participant Phone Number
- Phone Task
- Personal Profile
- Account Search
- Organization Search
- Performer Search

EVENT PROCESSOR / TASK ENGINE

Definition

Figure **14** is an illustration of the Event Processor **1400** in combination with other components of the system in accordance with on embodiment of the present invention. The Event Processor **1400** works behind the scenes of all claims applications to listen for significant events that have occurred in the life of various entities in the system like claims (but potentially many more like accounts or policies in the future). It determines what the response should be to each event and passes it onto the system component that will process it. The Event Processor is completely generic to any specific entity or event in the system and therefore enables automation based on an almost limitless number of events and responses that could be defined.

Figure **15** is an illustration of the Task Engine **1404** in accordance with one embodiment of the present invention. The Task Engine **1404** processes the most common set of event responses, those that need to generate tasks **1406** based on

events **1006** that have occurred. It compares the tasks that have been defined to the system to a set of claim criteria to tell which tasks should be added and which tasks should now be marked complete.

5   The only interface the user sees to these components is the task library **1500**, which allows task librarians **1502** to define the tasks and the rules that create them which are used by the Task Engine **1404**. Working with these components is almost entirely a function performed by specialists who understand the complexity of the rules involved in ensuring events **1006** and tasks **1406** are
10 handled properly.

The event processor **1400** also manages the communication and data synchronization between new claim components and LEGACY claim systems. This single point of contact effectively encapsulates the complex processes of
15 translation and notification of events between the two systems.

Value

The automated determination of event responses provides enormous benefits to system users by reducing the maintenance they have to perform in ensuring the correct disposition of claims. Users trigger events by the data they enter and the
20 system activities they perform, and the system automatically responds with appropriate automated activities like generating tasks.

The task generation rules defined in the Task Library provide an extremely flexible definition of claim handling processes limited only by the data available
25 in the system on which task creation rules can be based. Process changes can be implemented quickly by task librarians, and enforced through the Task Assistant.

Key Users

Although all claim personnel directly benefit from the functioning of the event processor and task assistant, only specially trained users control the processing of

these components. Task Librarians using the Task Library user interface handle the process of defining new tasks and the rules that trigger them in the Task Engine.

5   Operations personnel who ensure that all events are processed correctly and that the appropriate system resources are available to manage the throughput handle event processing.

Component Functionality

As shown in Figure **14**, the Event Processor **1400** utilizes a common queue **208**
10   of events **1006** that are populated by any component **1402** of the system to identify what events have occurred. Working this queue, the Event Processor determines the appropriate response for an event and provides information to other components that need to process them. The Event Processor does not process any events itself and maintains clear encapsulation of system
15   responsibilities. For example, an event that affects claim data is processed by the claim component.

The Task Engine **1404** follows a process of evaluating events **1006**, determining claim characteristics, and matching the claim's characteristics to tasks defined in
20   the Task Library **1500**.

The key user interface for the Task Engine **1404** is the Task Library **1500**. The Task Library **1500** maintains the templates that contain the fields and values with which tasks are established. A task template might contain statements like
25   "When event = litigation AND line of business = commercial auto, then..." Templates also identify what a tasks due date should be and how the task is enabled with other applications.

User Interfaces

- Search Task Template
- Search Triggering Templates
- Task Template Details

5

While various embodiments have been described above, it should be understood that they have been presented by way of example only, and not limitation. Thus, the breadth and scope of a preferred embodiment should not be limited by any of the above described exemplary embodiments, but should be defined only in

10 accordance with the following claims and their equivalents.